objc ↑↓

Advanced Swift

Updated for Swift 4

By Chris Eidhof, Ole Begemann, and Airspeed Velocity

Advanced Swift

Chris Eidhof

Ole Begemann

Airspeed Velocity

objc.io

Advanced Swift

Introduction

Advanced Swift is quite a bold title for a book, so perhaps we should start with what we mean by it.

When we began writing the first edition of this book, Swift was barely a year old. We did so before the beta of 2.0 was released — albeit tentatively, because we suspected the language would continue to evolve as it entered its second year. Few languages — perhaps no other language — have been adopted so rapidly by so many developers.

But that left people with unanswered questions. How do you write "idiomatic" Swift? Is there a correct way to do certain things? The standard library provided some clues, but even that has changed over time, dropping some conventions and adopting others. Over the past three years, Swift has evolved at a high pace, and it has become clearer what idiomatic Swift is.

To someone coming from another language, Swift can resemble everything you like about your language of choice. Low-level bit twiddling can look very similar to (and can be as performant as) C, but without many of the undefined behavior gotchas. The lightweight trailing closure syntax of `map` or `filter` will be familiar to Rubyists. Swift generics are similar to C++ templates, but with type constraints to ensure generic functions are correct at the time of definition rather than at the time of use. The flexibility of higher-order functions and operator overloading means you can write code that's similar in style to Haskell or F#. And the `@objc` and `dynamic` keywords allow you to use selectors and runtime dynamism in ways you would in Objective-C.

Given these resemblances, it's tempting to adopt the idioms of other languages. Case in point: Objective-C example projects can almost be mechanically ported to Swift. The same is true for Java or C# design patterns. And monad tutorials appeared to be everyone's favorite blog post topic in the first few months after Swift's introduction.

But then comes the frustration. Why can't we use protocol extensions with associated types like interfaces in Java? Why are arrays not covariant in the way we expect? Why can't we write "functor?" Sometimes the answer is because the part of Swift in question isn't yet implemented. But more often, it's either because there's a different Swift-like way to do what you want to do, or because the Swift feature you thought was like the equivalent in some other language is not quite what you think.

Swift is a complex language — most programming languages are. But it hides that complexity well. You can get up and running developing apps in Swift without needing to know about generics or overloading or the difference between static and dynamic dispatch. You may never need to call into a C library or write your own collection type, but after a while, we think you'll find it necessary to know about these things — either to improve your code's performance, to make it more elegant or expressive, or just to get certain things done.

Learning more about these features is what this book is about. We intend to answer many of the "How do I do this?" or "Why does Swift behave like that?" questions we've seen come up on various forums. Hopefully, once you've read our book, you'll have gone from being aware of the basics of the language to knowing about many advanced features and having a much better understanding of how Swift works. Being familiar with the material presented is probably necessary, if not sufficient, for calling yourself an advanced Swift programmer.

Who Is This Book For?

This book targets experienced (though not necessarily expert) programmers — such as existing Apple-platform developers, or those coming from other languages such as Java or C++ — who want to bring their knowledge of Swift to the same level as that of Objective-C or some other language. It's also suitable for new programmers who started on Swift, have grown familiar with the basics, and are looking to take things to the next level.

The book isn't meant to be an introduction to Swift; it assumes you're familiar with the syntax and structure of the language. If you want some good, compact coverage of the basics of Swift, the best source is the official Apple Swift book (available on iBooks or on Apple's website). If you're already a confident programmer, you could try reading our book and the Apple Swift book in parallel.

This is also not a book about programming for macOS or iOS devices. Of course, since Swift is currently mainly used on Apple platforms, we've tried to include examples of practical use, but we hope this book will be useful for non-Apple-platform programmers as well. The vast majority of the examples in the book should run unchanged on other operating systems. The ones that don't are either fundamentally tied to Apple's platforms (because they use iOS frameworks or rely on the Objective-C runtime) or only require minimal changes, such as replacing the BSD-specific function we use to generate random numbers with a Linux equivalent.

Themes

We've organized the book under the heading of basic concepts. There are in-depth chapters on some fundamental basic concepts like optionals or strings, and some deeper dives into topics like C interoperability. But throughout the book, hopefully a few themes regarding Swift emerge:

Swift is both a high- and low-level language. Swift allows you to write code similarly to Ruby and Python, with `map` and `reduce`, and to write your own higher-order functions easily. Swift also allows you to write fast code that compiles directly to native binaries with performance similar to code written in C.

What's exciting to us, and what's possibly the aspect of Swift we most admire, is that you're able to do both these things *at the same time*. Mapping a closure expression over an array compiles to the same assembly code as looping over a contiguous block of memory does.

However, there are some things you need to know about to make the most of this feature. For example, it will benefit you to have a strong grasp on how structs and classes differ, or an understanding of the difference between dynamic and static method dispatch. We'll cover topics such as these in more depth later on.

Swift is a multi-paradigm language. You can use it to write object-oriented code or pure functional code using immutable values, or you can write imperative C-like code using pointer arithmetic.

This is both a blessing and a curse. It's great, in that you have a lot of tools available to you, and you aren't forced into writing code one way. But it also exposes you to the risk of writing Java or C or Objective-C in Swift.

Swift still has access to most of the capabilities of Objective-C, including message sending, runtime type identification, and key-value observation. But Swift introduces many capabilities not available in Objective-C.

Erik Meijer, a well-known programming language expert, tweeted the following in October 2015:

> *At this point, @SwiftLang is probably a better, and more valuable, vehicle for learning functional programming than Haskell.*

Swift is a good introduction to a more functional style through its use of generics, protocols, value types, and closures. It's even possible to write operators that compose functions together. The early months of Swift brought many functional programming blog posts into the world. But since the release of Swift 2.0 and the introduction of protocol extensions, this trend has shifted.

Swift is very flexible. In the introduction to the book *On Lisp*, Paul Graham writes that:

> *Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called bottom-up design– changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together.*

Swift is a long way from Lisp. But still, we feel like Swift shares this characteristic of encouraging "bottom-up" programming — of making it easy to write very general reusable building blocks that you then combine into larger features, which you then use to solve your actual problem. Swift is particularly good at making these building blocks feel like primitives — like part of the language. A good demonstration of this is that the many features you might think of as fundamental building blocks, like optionals or basic operators, are actually defined in a library — the Swift standard library — rather than directly in the language. Trailing closures enable you to extend the language with features that feel like they're built in.

Swift code can be compact and concise while still being clear. Swift lends itself to relatively terse code. There's an underlying goal here, and it isn't to save on typing. The idea is to get to the point quicker and to make code readable by dropping a lot of the "ceremonial" boilerplate you often see in other languages that obscure rather than clarify the meaning of the code.

For example, type inference removes the clutter of type declarations that are obvious from the context. Semicolons and parentheses that add little or no value are gone. Generics and protocol extensions encourage you to avoid repeating yourself by packaging common operations into reusable functions. The goal is to write code that's readable at a glance.

At first, this can be off-putting. If you've never used functions like `map`, `filter`, and `reduce` before, they might look harder to read than a simple for loop. But our hope is that this is a short learning curve and that the reward is code that is more "obviously correct" at first glance.

Swift tries to be as safe as is practical, until you tell it not to be. This is unlike languages such as C and C++ (where you can be unsafe easily just by forgetting to do something), or like Haskell or Java (which are sometimes safe whether or not you like it).

Eric Lippert, one of the principal designers of C#, wrote about his 10 regrets of C#, including the lesson that:

sometimes you need to implement features that are only for experts who are building infrastructure; those features should be clearly marked as dangerous—not invitingly similar to features from other languages.

Eric was specifically referring to C#'s finalizers, which are similar to C++ destructors. But unlike destructors, they run at a nondeterministic time (perhaps never) at the behest of the garbage collector (and on the garbage collector's thread). However, Swift, being reference counted, *does* execute a class's `deinit` deterministically.

Swift embodies this sentiment in other ways. Undefined and unsafe behavior is avoided by default. For example, a variable can't be used until it's been initialized, and using out-of-bounds subscripts on an array will trap, as opposed to continuing with possibly garbage values.

There are a number of "unsafe" options available (such as the `unsafeBitcast` function, or the `UnsafeMutablePointer` type) for when you really need them. But with great power comes great undefined behavior. You can write the following:

```swift
var someArray = [1,2,3]
let uhOh = someArray.withUnsafeBufferPointer { ptr in
    // ptr is only valid within this block, but
    // there is nothing stopping you letting it
    // escape into the wild:
    return ptr
}
// Later...
print(uhOh[10])
```

It'll compile, but who knows what it'll do. However, you can't say nobody warned you.

Swift is an opinionated language. We as authors have strong opinions about the "right" way to write Swift. You'll see many of them in this book, sometimes expressed as if they're facts. But they're just, like, our opinions, man. Feel free to disagree! Swift is still a young language, and many things aren't settled. What's more is that many blog posts are flat-out wrong or outdated (including several ones we wrote, especially in the early days). Whatever you're reading, the most important thing is to try things out for yourself, check how they behave, and decide how you feel about them. Think critically, and beware of out-of-date information.

Swift continues to evolve. The period of major yearly syntax changes may be behind us, but important areas of the language are still new (strings), in flux (the generics system), or haven't been tackled yet (concurrency).

Terminology

> *'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean — neither more nor less.'*
>
> *— Through the Looking Glass, by Lewis Carroll*

Programmers throw around terms of art a lot. To avoid confusion, what follows are some definitions of terms we use throughout this book. Where possible, we're trying to adhere to the same usage as the official documentation, or sometimes a definition that's been widely adopted by the Swift community. Many of these definitions are covered in more detail in later chapters, so don't worry if not everything makes sense on first reading. If you're already familiar with all of these terms, it's still best to skim through to make sure your accepted meanings don't differ from ours.

In Swift, we make the distinctions between values, variables, references, and constants.

A **value** is immutable and forever — it never changes. For example, `1`, `true`, and `[1,2,3]` are all values. These are examples of **literals**, but values can also be generated at runtime. The number you get when you square the number five is a value.

When we assign a value to a name using `var x = [1,2]`, we're creating a **variable** named `x` that holds the value `[1,2]`. By changing `x`, e.g. by performing `x.append(3)`, we didn't change the original value. Rather, we replaced the value that `x` holds with the new value, `[1,2,3]` — at least *logically,* if not in the actual implementation (which might actually just tack a new entry on the back of some existing memory). We refer to this as **mutating** the variable.

We can declare **constant** variables (constants, for short) with `let` instead of `var`. Once a constant has been assigned a value, it can never be assigned a new value.

We also don't need to give a variable a value immediately. We can declare the variable first (`let x: Int`) and then later assign a value to it (`x = 1`). Swift, with its emphasis on safety, will check that all possible code paths lead to a variable being assigned a value before its value can be read. There's no concept of a variable having an as-yet-undefined value. Of course, if the variable was declared with `let`, it can only be assigned to once.

Structs and enums are **value types**. When you assign one struct variable to another, the two variables will then contain the same value. You can think of the contents as being copied, but it's more accurate to say that one variable was changed to contain the same value as the other.

A **reference** is a special kind of value: a value that "points to" another value. Because two references can refer to the same value, this introduces the possibility of that value getting mutated by two different parts of the program at once.

Classes are **reference types**. You can't hold an instance of a class (which we might occasionally call an **object** — a term fraught with troublesome overloading!) directly in a variable. Instead, you must hold a reference to it in a variable and access it via that reference.

Reference types have **identity** — you can check if two variables are referring to the exact same object by using `===`. You can also check if they're equal, assuming `==` is implemented for the relevant type. Two objects with different identity can still be equal.

Value types don't have identity. You can't check if a particular variable holds the "same" number 2 as another. You can only check if they both contain the value 2. `===` is really asking: "Do both these variables hold the same reference as their value?" In programming language literature, `==` is sometimes called *structural equality*, and `===` is called *pointer equality* or *reference equality*.

Class references aren't the only kind of reference in Swift. For example, there are also pointers, accessed through `withUnsafeMutablePointer` functions and the like. But classes are the simplest reference type to use, in part because their reference-like nature is partially hidden from you by syntactic sugar. You don't need to do any explicit "dereferencing" like you do with pointers in some other languages. (We'll cover the other kind of references in more detail in the chapter on interoperability.)

A variable that holds a reference can be declared with `let` — that is, the reference is constant. This means that the variable can never be changed to refer to something else. But — and this is important — it *doesn't* mean that the object it *refers to* can't be changed. So when referring to a variable as a constant, be careful — it's only constant in what it points to. It doesn't mean what it points to is constant. (Note: if those last few sentences sound like doublespeak, don't worry, as we cover this again in the chapter on structs and classes). Unfortunately, this means that when looking at a declaration of a variable with `let`, you can't tell at a glance whether or not what's being declared is completely immutable. Instead, you have to *know* whether it's holding a value type or a reference type.

We refer to types as having **value semantics** to distinguish a value type that performs a *deep copy*. This copy can occur eagerly (whenever a new variable is introduced) or lazily (whenever a variable gets mutated).

Here we hit another complication. If our struct contains reference types, the reference types won't automatically get copied upon assigning the struct to a new variable. Instead, the references themselves get copied. This is called a *shallow copy*.

For example, the `Data` struct in Foundation is a wrapper around the `NSData` reference type. However, the authors of the `Data` struct took extra steps to also perform a deep copy of the `NSData` object whenever the `Data` struct is mutated. They do this efficiently using a technique called copy-on-write, which we'll explain in the chapter on structs and classes. For now, it's important to know that this behavior doesn't come for free.

The collections in Swift are also wrapping reference types and use copy-on-write to efficiently provide value semantics. However, if the elements in a collection are references (for example, an array containing objects), the objects won't get copied. Instead, only the references get copied. This means that a Swift array only has value semantics if its elements have value semantics too.

Some classes are completely immutable — that is, they provide no methods for changing their internal state after they're created. This means that even though they're classes, they also have value semantics (because even if they're shared, they can never change). Be careful though — only `final` classes can be guaranteed not to be subclassed with added mutable state.

In Swift, functions are also values. You can assign a function to a variable, have an array of functions, and call the function held in a variable. Functions that take other functions as arguments (such as `map`, which takes a function to transform every element of a sequence) or return functions are referred to as **higher-order functions**.

Functions don't have to be declared at the top level — you can declare a function within another function or in a `do` or other scope. Functions defined within an outer scope, but passed out from it (say, as the returned value of a function), can "capture" local variables, in which case those local variables aren't destroyed when the local scope ends, and the function can hold state through them. This behavior is called "closing over" variables, and functions that do this are called **closures**.

Functions can be declared either with the `func` keyword or by using a shorthand `{ }` syntax called a **closure expression**. Sometimes this gets shortened to "closures," but don't let it give you the impression that only closure expressions can be closures. Functions declared with the `func` keyword are also closures when they close over external variables.

Functions are held by reference. This means assigning a function that has captured state to another variable doesn't copy that state; it shares it, similar to object references. What's more is that when two closures close over the same local variable, they both share that variable, so they share state. This can be quite surprising, and we'll discuss this more in the chapter on functions.

Functions defined inside a class or protocol are **methods**, and they have an implicit `self` parameter. Sometimes we call functions that aren't methods **free functions**. This is to distinguish them from methods.

A fully qualified function name in Swift includes not just the function's base name (the part before the parentheses), but also the argument labels. For example, the full name of the method for moving a collection index by a number of steps is `index(_:offsetBy:)`, indicating that this function takes two arguments (represented by the two colons), the first one of which has no label (represented by the underscore). We often omit the labels in the book if it's clear from the context what function we're referring to (the compiler allows you to do the same).

Free functions, and methods called on structs, are **statically dispatched**. This means the function that'll be called is known at compile time. It also means the compiler might be able to **inline** the function, i.e. not call the function at all, but instead replace it with the code the function would execute. The optimizer can also discard or simplify code that it can prove at compile time won't actually run.

Methods on classes or protocols might be **dynamically dispatched**. This means the compiler doesn't necessarily know at compile time which function will run. This dynamic behavior is done either by using vtables (similar to how Java or C++ dynamic dispatch work), or in the case of some `@objc` classes and protocols, by using selectors and `objc_msgSend`.

Subtyping and method **overriding** is one way of getting **polymorphic** behavior, i.e. behavior that varies depending on the types involved. A second way is function **overloading**, where a function is written multiple times for different types. (It's important not to mix up overriding and overloading, as they behave very differently.) A third way is via generics, where a function or method is written once to take any type that provides certain functions or methods, but the implementations of those functions can vary. Unlike method overriding, the results of function overloading and generics are known statically at compile time. We'll cover this more in the generics chapter.

Swift Style Guide

When writing this book, and when writing Swift code for our own projects, we try to stick to the following rules:

- For naming, clarity *at the point of use* is the most important consideration. Since APIs are used many more times than they're declared, their names should be optimized for how well they work at the call site. Familiarize yourself with the Swift API Design Guidelines and try to adhere to them in your own code.

- Clarity is often helped by conciseness, but brevity should never be a goal in and of itself.

- Always add documentation comments to functions — *especially* generic ones.

- Types start with `UpperCaseLetters`. Functions, variables, and enum cases start with `lowerCaseLetters`.

- Use type inference. Explicit but obvious types get in the way of readability.

- Don't use type inference in cases of ambiguity or when defining contracts (which is why, for example, funcs have an explicit return type).

- Default to structs unless you actually need a class-only feature or reference semantics.

- Mark classes as final unless you've explicitly designed them to be inheritable. If you want to use inheritance internally but not allow subclassing for external clients, mark a class public but not open.

- Use the trailing closure syntax, except when that closure is immediately followed by another opening brace.

- Use guard to exit functions early.

- Eschew force-unwraps and implicitly unwrapped optionals. They're occasionally useful, but needing them constantly is usually a sign something is wrong.

- Don't repeat yourself. If you find you've written a very similar piece of code more than a couple of times, extract it into a function. Consider making that function a protocol extension.

- Favor map and filter. But don't force it: use a for loop when it makes sense. The purpose of higher-order functions is to make code more readable. An obfuscated use of reduce when a simple for loop would be clearer defeats this purpose.

- Favor immutable variables: default to let unless you know you need mutation. But use mutation when it makes the code clearer or more efficient. Again, don't force it: a mutating method on a struct is often more idiomatic and efficient than returning a brand new struct.

- Swift generics tend to lead to very long function signatures. Unfortunately, we have yet to settle on a good way of breaking up long function declarations into multiple lines. We'll try to be consistent in how we do this in sample code.

- Leave off `self.` when you don't need it. In closure expressions, it's a clear signal that `self` is being captured by the closure.

- Write extensions on existing types and protocols, instead of free functions, whenever you can. This helps readability and discoverability.

One final note about our code samples throughout the book: to save space and focus on the essentials, we usually omit import statements that would be required to make the code compile. If you try out the code yourself and the compiler tells you it doesn't recognize a particular symbol, try adding an `import Foundation` or `import UIKit` statement.

Built-In Collections

Collections of elements are among the most important data types in any programming language. Good language support for different kinds of containers has a big impact on programmer productivity and happiness. Swift places special emphasis on sequences and collections — so much of the standard library is dedicated to this topic that we sometimes have the feeling it deals with little else. The resulting model is way more extensible than what you may be used to from other languages, but it's also quite complex.

In this chapter, we're going to take a look at the major collection types Swift ships with, with a focus on how to work with them effectively and idiomatically. In the next chapter, we'll climb up the abstraction ladder and see how the collection protocols in the standard library work.

Arrays

Arrays and Mutability

Arrays are the most common collections in Swift. An array is an ordered container of elements that all have the same type, with random access to each element. As an example, to create an array of numbers, we can write the following:

```
// The Fibonacci numbers
let fibs = [0, 1, 1, 2, 3, 5]
```

If we try to modify the array defined above (by using `append(_:)`, for example), we get a compile error. This is because the array is defined as a constant, using let. In many cases, this is exactly the right thing to do; it prevents us from accidentally changing the array. If we want the array to be a variable, we have to define it using var:

```
var mutableFibs = [0, 1, 1, 2, 3, 5]
```

Now we can easily append a single element or a sequence of elements:

```
mutableFibs.append(8)
mutableFibs.append(contentsOf: [13, 21])
mutableFibs // [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

There are a couple of benefits that come with making the distinction between var and let. Constants defined with let are easier to reason about because they're immutable. When you read a declaration like let fibs = ..., you know that the value of fibs will never change — it's enforced by the compiler. This helps greatly when reading through code. However, note that this is only true for types that have value semantics. A let variable to a class instance (i.e. a reference type) guarantees that the *reference* will never change, i.e. you can't assign another object to that variable. However, the object the reference points to *can* change. We'll go into more detail on these differences in the chapter on structs and classes.

Arrays, like all collection types in the standard library, have value semantics. When you assign an existing array to another variable, the array contents are copied over. For example, in the following code snippet, x is never modified:

```
var x = [1,2,3]
var y = x
y.append(4)
y // [1, 2, 3, 4]
x // [1, 2, 3]
```

The statement var y = x makes a copy of x, so appending 4 to y won't change x — the value of x will still be [1, 2, 3]. The same thing happens when you pass an array into a function; the function gets a local copy, and any changes it makes don't affect the caller.

Contrast this with the approach to mutability taken by NSArray in Foundation. NSArray has no mutating methods — to mutate an array, you need an NSMutableArray. But just because you have a non-mutating NSArray reference does *not* mean the array can't be mutated underneath you:

```
let a = NSMutableArray(array: [1,2,3])
// I don't want to be able to mutate b
let b: NSArray = a
// But it can still be mutated — via a
a.insert(4, at: 3)
b // ( 1, 2, 3, 4 )
```

The correct way to write this is to manually create a copy upon assignment:

```
let c = NSMutableArray(array: [1,2,3])
// I don't want to be able to mutate d
let d = c.copy() as! NSArray
c.insert(4, at: 3)
d // ( 1, 2, 3 )
```

In the example above, it's very clear that we need to make a copy — a is mutable, after all. However, when passing around arrays between methods and functions, this isn't always so easy to see.

In Swift, there's just one array type, and mutability is controlled by declaring with var instead of let. But there's no reference sharing — when you declare a second array with let, you're guaranteed it'll never change.

Making so many copies could be a performance problem, but in practice, all collection types in the Swift standard library are implemented using a technique called copy-on-write, which makes sure the data is only copied when necessary. So in our example, x and y shared internal storage up the point where y.append was called. In the chapter on structs and classes, we'll take a deeper look at value semantics, including how to implement copy-on-write for your own types.

Arrays and Optionals

Swift arrays provide all the usual operations you'd expect, like isEmpty and count. Arrays also allow for direct access of elements at a specific index through subscripting, like fibs[3]. Keep in mind that you need to make sure the index is within bounds before getting an element via subscript. Fetch the element at index 3, and you'd better be sure the array has at least four elements in it. Otherwise, your program will trap, i.e. abort with a fatal error.

The reason for this is mainly driven by how array indices are used. It's pretty rare in Swift to actually need to calculate an index:

- Want to iterate over the array?
  `for x in array`
- Want to iterate over all but the first element of an array?
  `for x in array.dropFirst()`
- Want to iterate over all but the last 5 elements?
  `for x in array.dropLast(5)`
- Want to number all the elements in an array?
  `for (num, element) in collection.enumerated()`
- Want to find the location of a specific element?
  `if let idx = array.index { someMatchingLogic($0) }`
- Want to transform all the elements in an array?
  `array.map { someTransformation($0) }`
- Want to fetch only the elements matching a specific criterion?
  `array.filter { someCriteria($0) }`

Another sign that Swift wants to discourage you from doing index math is the removal of traditional C-style for loops from the language in Swift 3. Manually fiddling with indices is a rich seam of bugs to mine, so it's often best avoided. And if it can't be, well, we'll see in the generics chapter that it's easy enough to write a new reusable general function that does what you need and in which you can wrap your carefully tested index calculations.

But sometimes you do have to use an index. And with array indices, the expectation is that when you do, you'll have thought very carefully about the logic behind the index calculation. So to have to unwrap the value of a subscript operation is probably overkill — it means you don't trust your code. But chances are you do trust your code, so you'll probably resort to force-unwrapping the result, because you *know* that the index must be valid. This is (a) annoying, and (b) a bad habit to get into. When force-unwrapping becomes routine, eventually you're going to slip up and force-unwrap something you don't mean to. So to avoid this habit becoming routine, arrays don't give you the option.

> *While a subscripting operation that responds to a invalid index with a controlled crash could arguably be called unsafe, that's only one aspect of safety. Subscripting is totally safe in regard to memory safety — the standard library collections always perform bounds checks to prevent unauthorized memory access with an out-of-bounds index.*

Other operations behave differently. The `first` and `last` properties have an optional type; they return `nil` if the array is empty. `first` is equivalent to `isEmpty ? nil : self[0]`. Similarly, the `removeLast` method will trap if you call it on an empty array, whereas `popLast` will only delete and return the last element if the array isn't empty, and will otherwise do nothing and return `nil`. Which one you'd want to use depends on your use case. When you're using the array as a stack, you'll probably always want to combine checking for `empty` and removing the last entry. On the other hand, if you already know through invariants whether or not the array is empty, dealing with the optional is fiddly.

We'll encounter these tradeoffs again later in this chapter when we talk about dictionaries. Additionally, there's an entire chapter dedicated to optionals.

Transforming Arrays

Map

It's common to need to perform a transformation on every value in an array. Every programmer has written similar code hundreds of times: create a new array, loop over all elements in an existing array, perform an operation on an element, and append the result of that operation to the new array. For example, the following code squares an array of integers:

```
var squared: [Int] = []
for fib in fibs {
    squared.append(fib * fib)
}
squared // [0, 1, 1, 4, 9, 25]
```

Swift arrays have a `map` method, adopted from the world of functional programming. Here's the exact same operation, using `map`:

```
let squares = fibs.map { fib in fib * fib }
squares // [0, 1, 1, 4, 9, 25]
```

This version has three main advantages. It's shorter, of course. There's also less room for error. But more importantly, it's clearer. All the clutter has been removed. Once you're used to seeing and using `map` everywhere, it acts as a signal — you see `map`, and you know immediately what's happening: a function is going to be applied to every element, returning a new array of the transformed elements.

The declaration of `squared` no longer needs to be made with `var`, because we aren't mutating it any longer — it'll be delivered out of the `map` fully formed, so we can declare `squares` with `let`, if appropriate. And because the type of the contents can be inferred from the function passed to `map`, `squares` no longer needs to be explicitly typed.

map isn't hard to write — it's just a question of wrapping up the boilerplate parts of the for loop into a generic function. Here's one possible implementation (though in Swift, it's actually an extension of Sequence, something we'll cover in the chapter on writing generic algorithms):

```
extension Array {
    func map<T>(_ transform: (Element) -> T) -> [T] {
        var result: [T] = []
        result.reserveCapacity(count)
        for x in self {
            result.append(transform(x))
        }
        return result
    }
}
```

Element is the generic placeholder for whatever type the array contains. And T is a new placeholder that can represent the result of the element transformation. The map function itself doesn't care what Element and T are; they can be anything at all. The concrete type T of the transformed elements is defined by the return type of the transform function the caller passes to map.

Really, the signature of this method should be func map<T>(_ transform: (Element) throws -> T) rethrows -> [T], indicating that map will forward any error the transformation function might throw to the caller. We'll cover this in detail in the errors chapter. We've left the error handling annotations out here for simplicity. If you'd like, you can check out the source code for Sequence.map in the Swift repository on GitHub.

Parameterizing Behavior with Functions

Even if you're already familiar with map, take a moment and consider the map code. What makes it so general yet so useful?

map manages to separate out the boilerplate — which doesn't vary from call to call — from the functionality that always varies: the logic of how exactly to transform each element. It does this through a parameter the caller supplies: the transformation function.

This pattern of parameterizing behavior is found throughout the standard library. There are more than a dozen separate functions that take a function that allows the caller to customize the key step:

- **map** and **flatMap** — how to transform an element
- **filter** — should an element be included?
- **reduce** — how to fold an element into an aggregate value
- **sequence** — what should the next element of the sequence be?
- **forEach** — what side effect to perform with an element
- **sort**, **lexicographicallyPrecedes**, and **partition** — in what order should two elements come?
- **index**, **first**, and **contains** — does this element match?
- **min** and **max** — which is the min/max of two elements?
- **elementsEqual** and **starts** — are two elements equivalent?
- **split** — is this element a separator?
- **prefix** — filter elements while a predicate returns true, then drop the rest (similar to `filter`, but with an early exit, and useful for infinite or lazily computed sequences)
- **drop** — drop elements until the predicate ceases to be true, and then return the rest (similar to `prefix`, but this returns the inverse)

The goal of all these functions is to get rid of the clutter of the uninteresting parts of the code, such as the creation of a new array, the `for` loop over the source data, and the like. Instead, the clutter is replaced with a single word that describes what's being done. This brings the important code – the logic the programmer wants to express – to the forefront.

Several of these functions have a default behavior. sort sorts elements in ascending order when they're comparable, unless you specify otherwise. contains can take a value to check for, so long as the elements are equatable. These defaults help make the code even more readable. Ascending order sort is natural, so the meaning of array.sort() is intuitive. array.index(of: "foo") is clearer than array.index { $0 == "foo" }.

But in every instance, these are just shorthand for the common cases. Elements don't have to be comparable or equatable, and you don't have to compare the whole element — you can sort an array of people by their ages (people.sort { $0.age < $1.age }) or check if the array contains anyone underage (people.contains { $0.age < 18 }). You can also compare some transformation of the element. For example, an admittedly inefficient case-insensitive sort could be performed via people.sort { $0.name.uppercased() < $1.name.uppercased() }.

There are other functions of similar usefulness that would also take a function to specify their behaviors but aren't in the standard library. You could easily define them yourself (and might like to try):

- **accumulate** — combine elements into an array of running values (like reduce, but returning an array of each interim combination)
- **all(matching:)** and **none(matching:)** — test if all or no elements in a sequence match a criterion (can be built with contains, with some carefully placed negation)
- **count(where:)** — count the number of elements that match (similar to filter, but without constructing an array)
- **indices(where:)** — return a list of indices matching a criteria (similar to index(where:), but doesn't stop on the first one)

Some of these we define elsewhere in the book.

You might find yourself writing something that fits a pattern more than a couple of times — something like this, where you search an array in reverse order for the first element that matches a certain condition:

```
let names = ["Paula", "Elena", "Zoe"]

var lastNameEndingInA: String?
for name in names.reversed() where name.hasSuffix("a") {
    lastNameEndingInA = name
    break
}
lastNameEndingInA // Optional("Elena")
```

If that's the case, consider writing a short extension to Sequence. The method last(where:) wraps this logic — we use a function argument to abstract over the part of our for loop that varies:

```
extension Sequence {
    func last(where predicate: (Element) -> Bool) -> Element? {
        for element in reversed() where predicate(element) {
            return element
        }
        return nil
    }
}
```

This then allows you to replace your for loop with the following:

```
let match = names.last { $0.hasSuffix("a") }
match // Optional("Elena")
```

This has all the same benefits we described for map. The example with last(where:) is more readable than the example with the for loop; even though the for loop is simple, you still have to run the loop through in your head, which is a small mental tax. Using last(where:) introduces less chance of error (such as accidentally forgetting to reverse the array), and it allows you to declare the result variable with let instead of var.

It also works nicely with guard — in all likelihood, you're going to terminate a flow early if the element isn't found:

```
guard let match = someSequence.last(where: { $0.passesTest() })
    else { return }
// Do something with match
```

We'll say more about extending collections and using functions later in the book.

Mutation and Stateful Closures

When iterating over an array, you could use map to perform side effects (e.g. inserting the elements into some lookup table). We don't recommend doing this. Take a look at the following:

```
array.map { item in
    table.insert(item)
}
```

This hides the side effect (the mutation of the lookup table) in a construct that looks like a transformation of the array. If you ever see something like the above, then it's a clear case for using a plain for loop instead of a function like map. The forEach method would also be more appropriate than map in this case, but it has its own issues. We'll look at forEach in a bit.

Performing side effects is different than deliberately giving the closure *local* state, which is a particularly useful technique, and it's what makes closures — functions that can capture and mutate variables outside their scope — so powerful a tool when combined with higher-order functions. For example, the accumulate function described above could be implemented with map and a stateful closure, like this:

```
extension Array {
    func accumulate<Result>(_ initialResult: Result,
        _ nextPartialResult: (Result, Element) -> Result) -> [Result]
    {
        var running = initialResult
        return map { next in
            running = nextPartialResult(running, next)
            return running
        }
    }
}
```

This creates a temporary variable to store the running value and then uses map to create an array of the running values as the computation progresses:

```
[1,2,3,4].accumulate(0, +) // [1, 3, 6, 10]
```

Note that this code assumes that map performs its transformation in order over the sequence. In the case of our map above, it does. But there are possible implementations that could transform the sequence out of order — for example, one that performs the transformation of the elements concurrently. The official standard library version of map doesn't specify whether or not it transforms the sequence in order, though it seems like a safe bet.

Filter

Another very common operation is to take an array and create a new array that only includes those elements that match a certain condition. The pattern of looping over an array and selecting the elements that match the given predicate is captured in the filter method:

```
let nums = [1,2,3,4,5,6,7,8,9,10]
nums.filter { num in num % 2 == 0 } // [2, 4, 6, 8, 10]
```

We can use Swift's shorthand notation for arguments of a closure expression to make this even shorter. Instead of naming the num argument, we can write the above code like this:

```
nums.filter { $0 % 2 == 0 } // [2, 4, 6, 8, 10]
```

For very short closures, this can be more readable. If the closure is more complicated, it's almost always a better idea to name the arguments explicitly, as we've done before. It's really a matter of personal taste — choose whichever is more readable at a glance. A good rule of thumb is this: if the closure fits neatly on one line, shorthand argument names are a good fit.

By combining map and filter, we can easily write a lot of operations on arrays without having to introduce a single intermediate variable, and the resulting code will become shorter and easier to read. For example, to find all squares under 100 that are even, we could map the range 0..<10, in order to square it, and then filter out all odd numbers:

```
(1..<10).map { $0 * $0 }.filter { $0 % 2 == 0 } // [4, 16, 36, 64]
```

The implementation of filter looks much the same as map:

```
extension Array {
    func filter(_ isIncluded: (Element) -> Bool) -> [Element] {
        var result: [Element] = []
        for x in self where isIncluded(x) {
            result.append(x)
        }
        return result
    }
}
```

For more on the where clause we use in the for loop, see the optionals chapter.

One quick performance tip: if you ever find yourself writing something like the following, stop!

```
bigArray.filter { someCondition }.count > 0
```

filter creates a brand new array and processes every element in the array. But this is unnecessary. This code only needs to check if one element matches — in which case, contains(where:) will do the job:

```
bigArray.contains { someCondition }
```

This is much faster for two reasons: it doesn't create a whole new array of the filtered elements just to count them, and it exits early, as soon as it finds the first match. Generally, only ever use filter if you want all the results.

Often you want to do something that can be done with contains, but it looks pretty ugly. For example, you can check if every element of a sequence matches a predicate using !sequence.contains { !condition }, but it’s much more readable to wrap this in a new function that has a more descriptive name:

```
extension Sequence {
    public func all(matching predicate: (Element) -> Bool) -> Bool {
        // Every element matches a predicate if no element doesn't match it:
        return !contains { !predicate($0) }
    }
}

let evenNums = nums.filter { $0 % 2 == 0 } // [2, 4, 6, 8, 10]
evenNums.all { $0 % 2 == 0 } // true
```

Reduce

Both map and filter take an array and produce a new, modified array. Sometimes, however, you want to combine all elements into a new value. For example, to sum up all the elements, we could write the following code:

```
let fibs = [0, 1, 1, 2, 3, 5]
var total = 0
for num in fibs {
    total = total + num
}
total // 12
```

The reduce method takes this pattern and abstracts two parts: the initial value (in this case, zero), and the function for combining the intermediate value (total) and the element (num). Using reduce, we can write the same example like this:

```
let sum = fibs.reduce(0) { total, num in total + num } // 12
```

Operators are functions too, so we could’ve also written the same example like this:

```
fibs.reduce(0, +) // 12
```

The output type of reduce doesn’t have to be the same as the element type. For example, if we want to convert a list of integers into a string, with each number followed by a space, we can do the following:

```
fibs.reduce("") { str, num in str + "\(num), " } // 0, 1, 1, 2, 3, 5,
```

Here's the implementation for reduce:

```
extension Array {
    func reduce<Result>(_ initialResult: Result,
        _ nextPartialResult: (Result, Element) -> Result) -> Result
    {
        var result = initialResult
        for x in self {
            result = nextPartialResult(result, x)
        }
        return result
    }
}
```

Another performance tip: reduce is very flexible, and it's common to see it used to build arrays and perform other operations. For example, you can implement map and filter using only reduce:

```
extension Array {
    func map2<T>(_ transform: (Element) -> T) -> [T] {
        return reduce([]) {
            $0 + [transform($1)]
        }
    }

    func filter2(_ isIncluded: (Element) -> Bool) -> [Element] {
        return reduce([]) {
            isIncluded($1) ? $0 + [$1] : $0
        }
    }
}
```

This is kind of beautiful and has the benefit of not needing those icky imperative for loops. But Swift isn't Haskell, and Swift arrays aren't lists. What's happening here is that every time, through the combine function, a brand new array is being created by appending the transformed or included element to the previous one. This means that both these implementations are $O(n^2)$, not $O(n)$ — as the length of the array increases, the time these functions take increases quadratically.

There's a second version of reduce, which has a different type. Specifically, the function for combining an intermediate result and an element now takes Result as an inout parameter:

```
public func reduce<Result>(into initialResult: Result,
    _ updateAccumulatingResult:
        (_ partialResult: inout Result, Element) throws -> ()
  ) rethrows -> Result
```

We discuss inout parameters in detail in the chapter on structs and classes, but for now, think of the inout Result parameter as a mutable parameter: we can modify it within the function. This allows us to write filter in a much more efficient way:

```
extension Array {
    func filter3(_ isIncluded: (Element) -> Bool) -> [Element] {
        return reduce(into: []) { result, element in
            if isIncluded(element) {
                result.append(element)
            }
        }
    }
}
```

When using inout, the compiler doesn't have to create a new array each time, so this version of filter is again $O(n)$. When the call to reduce(into:_:) is inlined by the compiler, the generated code is often the same as when using a for loop.

A Flattening Map

Sometimes, you want to map an array where the transformation function returns another array and not a single element.

For example, let's say we have a function, links, which reads a Markdown file and returns an array containing the URLs of all the links in the file. The function type looks like this:

```
func extractLinks(markdownFile: String) -> [URL]
```

If we have a bunch of Markdown files and want to extract the links from all files into a single array, we could try to write something like markdownFiles.map(extractLinks). But this returns an array of arrays containing the URLs: one array per file. Now you could just perform the map, get back an array of arrays, and then call joined to flatten the results into a single array:

```
let markdownFiles: [String] = // ...
let nestedLinks = markdownFiles.map(extractLinks)
let links = nestedLinks.joined()
```

The flatMap method combines these two operations into a single step. So markdownFiles.flatMap(extractLinks) returns all the URLs in an array of Markdown files as a single array.

The signature for flatMap is almost identical to map, except its transformation function returns an array. The implementation uses append(contentsOf:) instead of append(_:) to flatten the result array:

```
extension Array {
    func flatMap<T>(_ transform: (Element) -> [T]) -> [T] {
        var result: [T] = []
        for x in self {
            result.append(contentsOf: transform(x))
        }
        return result
    }
}
```

Another great use case for flatMap is combining elements from different arrays. To get all possible pairs of two arrays, flatMap over one array and then map over the other:

```
let suits = ["♠", "♥", "♣", "♦"]
let ranks = ["J","Q","K","A"]
let result = suits.flatMap { suit in
    ranks.map { rank in
        (suit, rank)
    }
}
/*
[("♠", "J"), ("♠", "Q"), ("♠", "K"), ("♠", "A"), ("♥", "J"), ("♥",
"Q"), ("♥", "K"), ("♥", "A"), ("♣", "J"), ("♣", "Q"), ("♣", "K"),
("♣", "A"), ("♦", "J"), ("♦", "Q"), ("♦", "K"), ("♦", "A")]
```

```
*/
```

Iteration using forEach

The final operation we'd like to discuss is forEach. It works almost like a for loop: the passed-in function is executed once for each element in the sequence. And unlike map, forEach doesn't return anything. Let's start by mechanically replacing a loop with forEach:

```
for element in [1,2,3] {
    print(element)
}

[1,2,3].forEach { element in
    print(element)
}
```

This isn't a big win, but it can be handy if the action you want to perform is a single function call on each element in a collection. Passing a function name to forEach instead of a closure expression can lead to clear and concise code. For example, if you're inside a view controller and want to add an array of subviews to the main view, you can just do theViews.forEach(view.addSubview).

However, there are some subtle differences between for loops and forEach. For instance, if a for loop has a return statement in it, rewriting it with forEach can significantly change the code's behavior. Consider the following example, which is written using a for loop with a where condition:

```
extension Array where Element: Equatable {
    func index(of element: Element) -> Int? {
        for idx in self.indices where self[idx] == element {
            return idx
        }
        return nil
    }
}
```

We can't directly replicate the where clause in the forEach construct, so we might (incorrectly) rewrite this using filter:

```
extension Array where Element: Equatable {
    func index_foreach(of element: Element) -> Int? {
        self.indices.filter { idx in
            self[idx] == element
        }.forEach { idx in
            return idx
        }
        return nil
    }
}
```

The return inside the forEach closure doesn't return out of the outer function; it only returns from the closure itself. In this particular case, we'd probably have found the bug because the compiler generates a warning that the argument to the return statement is unused, but you shouldn't rely on it finding every such issue.

Also, consider the following simple example:

```
(1..<10).forEach { number in
    print(number)
    if number > 2 { return }
}
```

It's not immediately obvious that this prints out all the numbers in the input range. The return statement isn't breaking the loop, rather it's returning from the closure.

In some situations, such as the addSubview example above, forEach can be nicer than a for loop. However, because of the non-obvious behavior with return, we recommend against most other uses of forEach. Just use a regular for loop instead.

Array Types

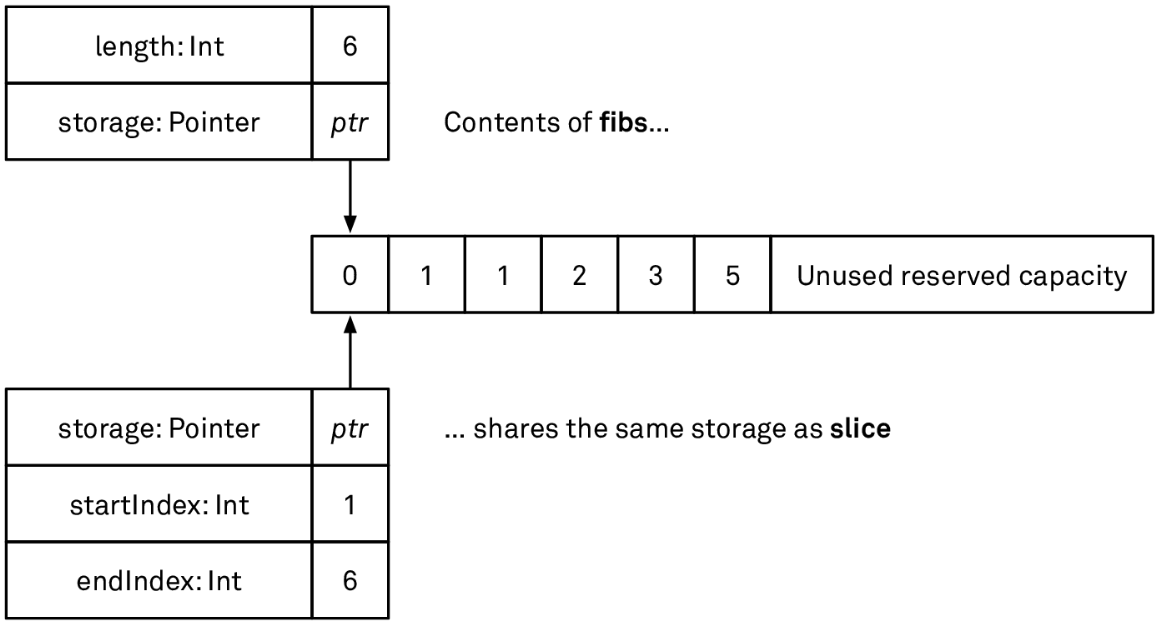
Slices

In addition to accessing a single element of an array by subscript (e.g. fibs[0]), we can also access a range of elements by subscript. For example, to get all but the first element of an array, we can do the following:

```
let slice = fibs[1...]
slice // [1, 1, 2, 3, 5]
type(of: slice) // ArraySlice<Int>
```

This gets us a slice of the array starting at the second element. The type of the result is ArraySlice, not Array. ArraySlice is a *view* on arrays. It's backed by the original array, yet it provides a view on just the slice. This makes certain the array doesn't need to get copied. The ArraySlice type has the same methods defined as Array does, so you can use a slice as if it were an array. If you do need to convert a slice into an array, you can just construct a new array out of the slice:

```
let newArray = Array(slice)
type(of: newArray) // Array<Int>
```



Array Slices

Bridging

Swift arrays can bridge to Objective-C. They can also be used with C, but we'll cover that in a later chapter. Because NSArray can only hold objects, the compiler and runtime will automatically wrap incompatible values (for example, Swift enums) in an opaque box object. A number of value types (such as Int, Bool, and String, but also Dictionary and Set) are bridged automatically to their Objective-C counterparts.

> *A universal bridging mechanism for all Swift types to Objective-C doesn't just make working with arrays more pleasant. It also applies to other collections, like dictionaries and sets, and it opens up a lot of potential for future enhancements to the interoperability between Swift and Objective-C. For example, a future Swift version might allow Swift value types to conform to @objc protocols.*

Dictionaries

Another key data structure is Dictionary. A dictionary contains keys with corresponding values; duplicate keys aren't supported. Retrieving a value by its key takes constant time on average, whereas searching an array for a particular element grows linearly with the array's size. Unlike arrays, dictionaries aren't ordered. The order in which pairs are enumerated in a for loop is undefined.

In the following example, we use a dictionary as the model data for a fictional settings screen in a smartphone app. The screen consists of a list of settings, and each individual setting has a name (the keys in our dictionary) and a value. A value can be one of several data types, such as text, numbers, or booleans. We use an enum with associated values to model this:

```
enum Setting {
    case text(String)
    case int(Int)
    case bool(Bool)
}

let defaultSettings: [String:Setting] = [
    "Airplane Mode": .bool(false),
    "Name": .text("My iPhone"),
]
defaultSettings["Name"] // Optional(Setting.text("My iPhone"))
```

We use subscripting to get the value of a setting. Dictionary lookup always returns an *optional value* — when the specified key doesn't exist, it returns nil. Contrast this with arrays, which respond to an out-of-bounds access by crashing the program.

The rationale for this difference is that array indices and dictionary keys are used very differently. We've already seen that it's quite rare that you actually need to work with array indices directly. And if you do, an array index is usually directly derived from the array in some way (e.g. from a range like 0..<array.count); thus, using an invalid index is a programmer error. On the other hand, it's very common for dictionary keys to come from some source other than the dictionary itself.

Unlike arrays, dictionaries are also sparse. The existence of the value under the key "name" doesn't tell you anything about whether or not the key "address" also exists.

Mutation

Just like with arrays, dictionaries defined using let are immutable: no entries can be added, removed, or changed. And just like with arrays, we can define a mutable variant using var. To remove a value from a dictionary, we can either set it to nil using subscripting or call removeValue(forKey:). The latter additionally returns the deleted value, or nil if the key didn't exist. If we want to take an immutable dictionary and make changes to it, we have to make a copy:

```
var userSettings = defaultSettings
userSettings["Name"] = .text("Jared's iPhone")
userSettings["Do Not Disturb"] = .bool(true)
```

Note that, again, the value of defaultSettings didn't change. As with key removal, an alternative to updating via subscript is the updateValue(_:forKey:) method, which returns the previous value (if any):

```
let oldName = userSettings
    .updateValue(.text("Jane's iPhone"), forKey: "Name")
userSettings["Name"] // Optional(Setting.text("Jane\'s iPhone"))
oldName // Optional(Setting.text("Jared\'s iPhone"))
```

Some Useful Dictionary Methods

What if we wanted to combine the default settings dictionary with any custom settings the user has changed? Custom settings should override defaults, but the resulting dictionary should still include default values for any keys that haven't been customized. Essentially, we want to merge two dictionaries, where the dictionary that's being merged in overwrites duplicate keys.

Dictionary has a merge(_:uniquingKeysWith:) method, which takes the key-value pairs to be merged in and a function that specifies how to combine two values with the same key. We can use this to merge one dictionary into another, as shown in the following example:

```
var settings = defaultSettings
let overriddenSettings: [String:Setting] = ["Name": .text("Jane's iPhone")]
settings.merge(overriddenSettings, uniquingKeysWith: { $1 })
settings
// ["Name": Setting.text("Jane\'s iPhone"), "Airplane Mode": Setting.bool(false)]
```

In the example above, we used { $1 } as the policy for combining two values. In other words, in case a key exists in both settings and overriddenSettings, we use the value from overriddenSettings.

We can also construct a new dictionary out of a sequence of (Key,Value) pairs. If we guarantee that the keys are unique, we can use Dictionary(uniqueKeysWithValues:). However, in case we have a sequence where a key can exist multiple times, we need to provide a function to combine two values for the same keys, just like above. For example, to compute how often elements appear in a sequence, we can map over each element, combine it with a 1, and then create a dictionary out of the resulting element-frequency pairs. If we encounter two values for the same key (in other words, if we saw the same element more than once), we simply add the frequencies up using +:

```
extension Sequence where Element: Hashable {
    var frequencies: [Element:Int] {
        let frequencyPairs = self.map { ($0, 1) }
        return Dictionary(frequencyPairs, uniquingKeysWith: +)
    }
}
let frequencies = "hello".frequencies // ["e": 1, "o": 1, "l": 2, "h": 1]
frequencies.filter { $0.value > 1 } // ["l": 2]
```

Another useful method is a map over the dictionary's values. Because Dictionary is a Sequence, it already has a map method that produces an array. However, sometimes we want to keep the dictionary structure intact and only transform its values. The mapValues method does just this:

```
let settingsAsStrings = settings.mapValues { setting -> String in
    switch setting {
    case .text(let text): return text
    case .int(let number): return String(number)
    case .bool(let value): return String(value)
    }
}
settingsAsStrings // ["Name": "Jane\'s iPhone", "Airplane Mode": "false"]
```

Hashable Requirement

Dictionaries are hash tables. The dictionary assigns each key a position in its underlying storage array based on the key's hashValue. This is why Dictionary requires its Key type to conform to the Hashable protocol. All the basic data types in the standard library already do, including strings, integers, floating-point, and Boolean values. Enumerations without associated values also get automatic Hashable conformance for free.

If you want to use your own custom types as dictionary keys, you must add `Hashable` conformance manually. This requires an implementation of the `hashValue` property and, because `Hashable` extends `Equatable`, an overload of the `==` operator function for your type. Your implementation must hold an important invariant: two instances that are equal (as defined by your `==` implementation) *must* have the same hash value. The reverse isn't true: two instances with the same hash value don't necessarily compare equally. This makes sense, considering that there's only a finite number of distinct hash values, while many hashable types (like strings) have essentially infinite cardinality.

The potential for duplicate hash values means that `Dictionary` must be able to handle collisions. Nevertheless, a good hash function should strive for a minimal number of collisions in order to preserve the collection's performance characteristics, i.e. the hash function should produce a uniform distribution over the full integer range. In the extreme case where your implementation returns the same hash value (e.g. zero) for every instance, a dictionary's lookup performance degrades to $O(n)$.

The second characteristic of a good hash function is that it's fast. Keep in mind that the hash value is computed every time a key is inserted, removed, or looked up. If your `hashValue` implementation takes too much time, it might eat up any gains you got from the $O(1)$ complexity.

Writing a good hash function that meets these requirements isn't easy. Fortunately, soon we won't have to do this ourselves in most situations, as there's an accepted proposal for synthesizing `Equatable` and `Hashable` conformance. Once this feature is fully implemented and merged, we can instruct the compiler to automatically generate both `Equatable` and `Hashable` conformance for our custom types.

Finally, be extra careful when you use types that don't have value semantics (e.g. mutable objects) as dictionary keys. If you mutate an object after using it as a dictionary key in a way that changes its hash value and/or equality, you'll not be able to find it again in the dictionary. The dictionary now stores the object in the wrong slot, effectively corrupting its internal storage. This isn't a problem with value types because the key in the dictionary doesn't share your copy's storage and therefore can't be mutated from the outside.

Sets

The third major collection type in the standard library is Set. A set is an unordered collection of elements, with each element appearing only once. You can essentially think of a set as a dictionary that only stores keys and no values. Like Dictionary, Set is implemented with a hash table and has similar performance characteristics and requirements. Testing a value for membership in the set is a constant-time operation, and set elements must be Hashable, just like dictionary keys.

Use a set instead of an array when you need to test efficiently for membership (an $O(n)$ operation for arrays) and the order of the elements is not important, or when you need to ensure that a collection contains no duplicates.

Set conforms to the ExpressibleByArrayLiteral protocol, which means that we can initialize it with an array literal like this:

```
let naturals: Set = [1, 2, 3, 2]
naturals // [2, 3, 1]
naturals.contains(3) // true
naturals.contains(0) // false
```

Note that the number 2 appears only once in the set; the duplicate never even gets inserted.

Like all collections, sets support the common operations we've already seen: you can iterate over the elements in a for loop, map or filter them, and do all other sorts of things.

Set Algebra

As the name implies, Set is closely related to the mathematical concept of a set; it supports all common set operations you learned in math class. For example, we can *subtract* one set from another:

```
let iPods: Set = ["iPod touch", "iPod nano", "iPod mini",
    "iPod shuffle", "iPod Classic"]
let discontinuedIPods: Set = ["iPod mini", "iPod Classic",
    "iPod nano", "iPod shuffle"]
let currentIPods = iPods.subtracting(discontinuedIPods) // ["iPod touch"]
```

We can also form the *intersection* of two sets, i.e. find all elements that are in both:

```
let touchscreen: Set = ["iPhone", "iPad", "iPod touch", "iPod nano"]
let iPodsWithTouch = iPods.intersection(touchscreen)
// ["iPod touch", "iPod nano"]
```

Or, we can form the *union* of two sets, i.e. combine them into one (removing duplicates, of course):

```
var discontinued: Set = ["iBook", "Powerbook", "Power Mac"]
discontinued.formUnion(discontinuedIPods)
discontinued
/*
["iBook", "Powerbook", "Power Mac", "iPod Classic", "iPod mini",
"iPod shuffle", "iPod nano"]
*/
```

Here, we used the mutating variant formUnion to mutate the original set (which, as a result, must be declared with var). Almost all set operations have both non-mutating and mutating forms, the latter beginning with form.... For even more set operations, check out the SetAlgebra protocol.

Index Sets and Character Sets

Set and OptionSet are the only types in the standard library that conform to SetAlgebra, but the protocol is also adopted by two interesting types in Foundation: IndexSet and CharacterSet. Both of these date back to a time long before Swift was a thing. The way these and other Objective-C classes are now bridged into Swift as fully featured value types — adopting common standard library protocols in the process — is great because they'll instantly feel familiar to Swift developers.

IndexSet represents a set of positive integer values. You can, of course, do this with a Set<Int>, but IndexSet is more storage efficient because it uses a list of ranges internally. Say you have a table view with 1,000 elements and you want to use a set to manage the indices of the rows the user has selected. A Set<Int> needs to store up to 1,000 elements, depending on how many rows are selected. An IndexSet, on the other hand, stores continuous ranges, so a selection of the first 500 rows in the table only takes two integers to store (the selection's lower and upper bounds).

However, as a user of an IndexSet, you don't have to worry about the internal structure, as it's completely hidden behind the familiar SetAlgebra and Collection interfaces. (Unless you want to work on the ranges directly, that is. IndexSet exposes a view to them via its rangeView property, which itself is a collection.) For example, you can add a few ranges to an index set and then map over the indices as if they were individual members:

```
var indices = IndexSet()
indices.insert(integersIn: 1..<5)
indices.insert(integersIn: 11..<15)
let evenIndices = indices.filter { $0 % 2 == 0 } // [2, 4, 12, 14]
```

CharacterSet is an equally efficient way to store a set of Unicode code points. It's often used to check if a particular string only contains characters from a specific character subset, such as alphanumerics or decimalDigits. However, unlike IndexSet, CharacterSet isn't a collection. The name CharacterSet, imported from Objective-C, is also unfortunate in Swift because CharacterSet isn't compatible with

Swift's Character type. A better name would be UnicodeScalarSet. We'll talk a bit more about CharacterSet in the chapter on strings.

Using Sets Inside Closures

Dictionaries and sets can be very handy data structures to use inside your functions, even when you're not exposing them to the caller. For example, if we want to write an extension on Sequence to retrieve all unique elements in the sequence, we could easily put the elements in a set and return its contents. However, that won't be *stable*: because a set has no defined order, the input elements might get reordered in the result. To fix this, we can write an extension that maintains the order by using an internal Set for bookkeeping:

```
extension Sequence where Element: Hashable {
    func unique() -> [Element] {
        var seen: Set<Element> = []
        return filter { element in
            if seen.contains(element) {
                return false
            } else {
                seen.insert(element)
                return true
            }
        }
    }
}

[1,2,3,12,1,3,4,5,6,4,6].unique() // [1, 2, 3, 12, 4, 5, 6]
```

The method above allows us to find all unique elements in a sequence while still maintaining the original order (with the constraint that the elements must be Hashable). Inside the closure we pass to filter, we refer to the variable seen that we defined outside the closure, thus maintaining state over multiple iterations of the closure. In the chapter on functions, we'll look at this technique in more detail.

Ranges

A range is an interval of values, defined by its lower and upper bounds. You create ranges with the two range operators: ..< for half-open ranges that don't include their upper bound, and ... for closed ranges that include both bounds:

```
// 0 to 9, 10 is not included
let singleDigitNumbers = 0..<10
Array(singleDigitNumbers) // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
// "z" is included
let lowercaseLetters = Character("a")...Character("z")
```

There are also prefix and postfix variants of these operators, used to express one-sided ranges:

```
let fromZero = 0...
let upToZ = ..<Character("z")
```

There are eight distinct concrete types that represent ranges, and each type captures different constraints on the value. The two most essential types are Range (a half-open range, created using ..<) and ClosedRange (created using ...). Both have a generic Bound parameter: the only requirement is that Bound must be Comparable. For example, the lowercaseLetters expression above is of type ClosedRange<Character>. Somewhat surprisingly, we can't iterate over Range and ClosedRange, but we can check whether elements are contained within the range:

```
singleDigitNumbers.contains(9) // true
lowercaseLetters.overlaps("c"..<"f") // true
```

(The answer why iterating over characters isn't as straightforward as it would seem has to do with Unicode, and we'll cover it at length in the chapter on strings.)

Half-open and closed ranges both have a place:

- Only a **half-open range** can represent an **empty interval** (when the lower and upper bounds are equal, as in 5..<5).

- Only a **closed range** can contain the **maximum value** its element type can represent (e.g. 0...Int.max). A half-open range always requires at least one representable value that's greater than the highest value in the range.

Countable Ranges

Ranges seem like a natural fit to be sequences or collections, so it may surprise you that Range and ClosedRange are neither. Some ranges *are* sequences, however, otherwise iterating over a range of integers in a for loop wouldn't work:

```
for i in 0..<10 {
    print("\(i)", terminator: " ")
} // 0 1 2 3 4 5 6 7 8 9
```

What's going on here? If you inspect the type of the expression 0..<10, you'll discover it's CountableRange<Int>. A CountableRange is just like a Range, with the additional constraint that its element type conforms to Strideable (with integer steps). Swift calls these more capable ranges *countable* because only they can be iterated over. Valid bounds for countable ranges include integer and pointer types — but not floating-point types, because of the integer constraint on the type's Stride. If you need to iterate over consecutive floating-point values, you can use the stride(from:to:by) and stride(from:through:by) functions to create such a sequence. The Strideable constraint enables CountableRange and CountableClosedRange to conform to RandomAccessCollection, so we can iterate over them.

The four basic range types we've discussed thus far can be classified in a two-by-two matrix, as follows:

	Half-open range	**Closed range**
Elements are Comparable	Range	ClosedRange
Elements are Strideable (with integer steps)	CountableRange	CountableClosedRange

The columns of the matrix correspond to the two range operators we saw above, which create a [Countable]Range (half-open) or a [Countable]ClosedRange (closed), respectively.

The rows in the table distinguish between "normal" ranges with an element type that only conforms to the Comparable protocol (which is the minimum requirement), and ranges over types that are Strideable *and* use integer steps between elements. Only the latter ranges are collections, inheriting all the powerful functionality we'll see in the next chapter.

Partial Ranges

Partial ranges are constructed by using ... or ..< as a prefix or a postfix operator. For example, 0... means the range starting at zero. These ranges are called partial because they're still missing one of their bounds. There are four different kinds:

```
let fromA: PartialRangeFrom<Character> = Character("a")...
let throughZ: PartialRangeThrough<Character> = ...Character("z")
let upto10: PartialRangeUpTo<Int> = ..<10
let fromFive: CountablePartialRangeFrom<Int> = 5...
```

There's only one countable variant: CountablePartialRangeFrom. It's the only partial range we can iterate over. Iteration works by starting with the lowerBound and repeatedly calling advanced(by: 1). If you use such a range in a for loop, you must take care to add a break condition lest you end up in an infinite loop (or crash when the counter overflows). PartialRangeFrom can't be iterated over because its Bound is not Strideable, and both PartialRangeThrough and PartialRangeUpTo are missing a lower bound.

Range Expressions

All eight range types conform to the RangeExpression protocol. The protocol itself is small enough to print in this book. First of all, it allows you to ask whether an element is contained within the range. Second, given a collection, it can compute a fully specified Range for you:

```
public protocol RangeExpression {
    associatedtype Bound: Comparable
    func contains(_ element: Bound) -> Bool
    func relative<C: _Indexable>(to collection: C) -> Range<Bound>
        where C.Index == Bound
}
```

For partial ranges with a missing lower bound, the relative(to:) method adds the collection's startIndex as the lower bound. For partial ranges with a missing upper bound, it'll use the collection's endIndex. Partial ranges enable a very compact syntax for slicing collections:

```
let arr = [1,2,3,4]
arr[2...] // [3, 4]
arr[..<1] // [1]
arr[1...2] // [2, 3]
```

This works because the corresponding subscript declaration in the Collection protocol takes a RangeExpression rather than one of the eight concrete range types. You can even omit both bounds to get a slice of the entire collection:

```
arr[...] // [1, 2, 3, 4]
type(of: arr) // Array<Int>
```

(This is implemented as a special case in the standard library in Swift 4.0. Such an *unbounded range* isn't a valid RangeExpression yet, but it should become one eventually.)

Ranges and Conditional Conformance

The standard library currently has to have separate types for countable ranges: CountableRange, CountableClosedRange, and CountablePartialRangeFrom. Ideally, these wouldn't be distinct types, but rather extensions on Range, ClosedRange, and PartialRangeFrom that declare Collection conformance on the condition that the generic parameters meet the required constraints. We'll talk a lot more about this in the next chapter, but the code might look like this:

```
// Invalid in Swift 4
extension Range: RandomAccessCollection
    where Bound: Strideable, Bound.Stride: SignedInteger
{
    // Implement RandomAccessCollection
}
```

Alas, Swift 4's type system can't express this idea, so separate types are needed. Support for conditional conformance is expected for Swift 5, and CountableRange, CountableClosedRange, and CountablePartialRangeFrom will then be deprecated or removed from the standard library.

The distinction between the half-open Range and the closed ClosedRange will likely remain, and it can sometimes make working with ranges harder than it should intuitively be. Say you have a function that takes a Range<Character> and you want to pass it the closed character range we created above. You may be surprised to find out it's not possible! Inexplicably, there appears to be no way to convert a ClosedRange into a Range. But why? Well, to turn a closed range into an equivalent half-open range, you'd have to find the element that comes after the original range's upper bound. And that's simply not possible unless the element is Strideable, which is only guaranteed for countable ranges.

This means the caller of such a function will have to provide the correct type. If the function expects a Range, you can't use the ... operator to create it. We're not certain how big of a limitation this is in practice, since most ranges are likely integer based, but it's definitely unintuitive.

If possible, try copying the standard library's approach and make your own functions take a RangeExpression rather than a concrete type. It's not always possible because the protocol doesn't give you access to the range's bounds unless you're in the context of a collection, but if it is, you'll give consumers of your APIs much more freedom to pass in any kind of range expression they like.

Recap

In this chapter, we saw a number of different collections: arrays, dictionaries, sets, index sets, and ranges. We also looked at a number of methods that each of these collections have, and in the next chapter, we'll see that most of those methods are defined on the `Sequence` protocol. We saw how Swift's built-in collections allow you to control mutability through `let` and `var`, and we also saw how to make sense of all the different `Range` types.

Collection Protocols

We saw in the previous chapter that Array, Dictionary, and Set don't exist in a vacuum. They're all implemented on top of a rich set of abstractions for processing series of elements that's provided by the Swift standard library. This chapter is all about the Sequence and Collection protocols, which form the cornerstones of this model. We'll cover how these protocols work, why they work the way they do, and how you can write your own sequences and collections.

Sequences

The Sequence protocol stands at the base of the hierarchy. A sequence is a series of values of the same type that lets you iterate over the values. The most common way to traverse a sequence is a for loop:

```
for element in someSequence {
    doSomething(with: element)
}
```

This seemingly simple capability of stepping over elements forms the foundation for a large number of useful operations Sequence provides to adopters of the protocol. We already mentioned many of them in the previous chapter. Whenever you come up with a common operation that depends on sequential access to a series of values, you should consider implementing it on top of Sequence, too. We'll see many examples of how to do this throughout this chapter and the rest of the book.

The requirements for a type to conform to Sequence are fairly small. All it must do is provide a makeIterator() method that returns an *iterator*:

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    func makeIterator() -> Iterator
    // ...
}
```

The only thing we learn from this (simplified) definition of Sequence is that it's a type that knows how to make an iterator. So let's first take a closer look at iterators.

Iterators

Sequences provide access to their elements by creating an iterator. The iterator produces the values of the sequence one at a time and keeps track of its iteration state as it traverses through the sequence. The only method defined in the IteratorProtocol protocol is next(), which must return the next element in the sequence on each subsequent call, or nil when the sequence is exhausted:

```
protocol IteratorProtocol {
    associatedtype Element
    mutating func next() -> Element?
}
```

The associated Element type specifies the type of the values the iterator produces. For example, the element type of the iterator for String is Character. By extension, the iterator also defines its sequence's element type. (We'll talk more about protocols with associated types later in this chapter and in the chapter on protocols):

```
public protocol Sequence {
    associatedtype Element
    associatedtype Iterator: IteratorProtocol
        where Iterator.Element == Element

    // ...
}
```

You normally only have to care about iterators when you implement one for a custom sequence type. Other than that, you rarely need to use iterators directly, because a for loop is the idiomatic way to traverse a sequence. In fact, this is what a for loop does under the hood: the compiler creates a fresh iterator for the sequence and calls next on that iterator repeatedly, until nil is returned. The for loop example we showed above is essentially shorthand for the following:

```
var iterator = someSequence.makeIterator()
while let element = iterator.next() {
    doSomething(with: element)
}
```

Iterators are single-pass constructs; they can only be advanced, and never reversed or reset. While most iterators will produce a finite number of elements and eventually return nil from next(), nothing stops you from vending an infinite series that never ends. As a matter of fact, the simplest iterator imaginable — short of one that immediately returns nil — is one that just returns the same value over and over again:

```
struct ConstantIterator: IteratorProtocol {
    typealias Element = Int
    mutating func next() -> Int? {
        return 1
    }
}
```

The explicit typealias for Element is optional (but often useful for documentation purposes, especially in larger protocols). If we omit it, the compiler infers the concrete type of Element from the return type of next():

```
struct ConstantIterator: IteratorProtocol {
    mutating func next() -> Int? {
        return 1
    }
}
```

Notice that the next() method is declared as mutating. This isn't strictly necessary in this simplistic example because our iterator has no mutable state. In practice, though, iterators are inherently stateful. Almost any useful iterator requires mutable state to keep track of its position in the sequence.

We can create a new instance of ConstantIterator and loop over the sequence it produces in a while loop, printing an endless stream of ones:

```
var iterator = ConstantIterator()
while let x = iterator.next() {
    print(x)
}
```

Let's look at a more elaborate example. FibsIterator produces the Fibonacci sequence. It keeps track of the current position in the sequence by storing the upcoming two numbers. The next method then returns the first number and updates the state for the following call. Like the previous example, this iterator also produces an "infinite" stream; it keeps generating numbers until it reaches integer overflow, and then the program crashes:

```
struct FibsIterator: IteratorProtocol {
    var state = (0, 1)
    mutating func next() -> Int? {
        let upcomingNumber = state.0
        state = (state.1, state.0 + state.1)
        return upcomingNumber
    }
}
```

Conforming to Sequence

An example of an iterator that produces a finite sequence is the following PrefixIterator, which generates all prefixes of a string (including the string itself). It starts at the beginning of the string, and with each call of next, increments the slice of the string it returns by one character until it reaches the end:

```
struct PrefixIterator: IteratorProtocol {
    let string: String
    var offset: String.Index

    init(string: String) {
        self.string = string
        offset = string.startIndex
    }
    mutating func next() -> Substring? {
        guard offset < string.endIndex else { return nil }
        offset = string.index(after: offset)
        return string[..<offset]
    }
}
```

(string[..<offset] is a slicing operation that returns the substring between the start and the offset — we saw the partial range notation in the previous chapter, and we'll talk more about slicing later.)

With PrefixIterator in place, defining the accompanying PrefixSequence type is easy. Again, it isn't necessary to specify the associated Iterator or Element types explicitly because the compiler can infer them from the return type of the makeIterator method:

```
struct PrefixSequence: Sequence {
    let string: String
    func makeIterator() -> PrefixIterator {
        return PrefixIterator(string: string)
    }
}
```

Now we can use a for loop to iterate over all the prefixes:

```
for prefix in PrefixSequence(string: "Hello") {
    print(prefix)
}
/*
H
He
Hel
Hell
Hello
*/
```

Or we can perform any other operation provided by Sequence:

```
PrefixSequence(string: "Hello").map { $0.uppercased() }
// ["H", "HE", "HEL", "HELL", "HELLO"]
```

We can create sequences for ConstantIterator and FibsIterator in the same way. We're not showing them here, but you may want to try this yourself. Just keep in mind that these iterators create infinite sequences. Use a construct like for i in fibsSequence.prefix(10) to slice off a finite piece.

Iterators and Value Semantics

The iterators we've seen thus far all have value semantics. If you make a copy of one, the iterator's entire state will be copied, and the two instances will behave independently of one other, as you'd expect. Most iterators in the standard library also have value semantics, but there are exceptions.

To illustrate the difference between value and reference semantics, we first take a look at StrideToIterator. It's the underlying iterator for the sequence that's returned from the stride(from:to:by:) function. Let's create a StrideToIterator and call next a couple of times:

```
// A sequence from 0 to 9
let seq = stride(from: 0, to: 10, by: 1)
var i1 = seq.makeIterator()
i1.next() // Optional(0)
i1.next() // Optional(1)
```

i1 is now ready to return 2. Now, say you make a copy of it:

```
var i2 = i1
```

Both the original and the copy are now separate and independent, and both return 2 when you call next:

```
i1.next() // Optional(2)
i1.next() // Optional(3)
i2.next() // Optional(2)
i2.next() // Optional(3)
```

This is because StrideToIterator, a pretty simple struct whose implementation is not too dissimilar from our Fibonacci iterator above, has value semantics.

Now let's look at an iterator that doesn't have value semantics. AnyIterator is an iterator that wraps another iterator, thus "erasing" the base iterator's concrete type. An example where this might be useful is if you want to hide the concrete type of a complex iterator that would expose implementation details in your public API. The way AnyIterator does this is by wrapping the base iterator in an internal box object, which is a reference type. (If you want to learn exactly how this works, check out the section on type erasure in the protocols chapter.)

To see why this is relevant, we create an AnyIterator that wraps i1, and then we make a copy:

```
var i3 = AnyIterator(i1)
var i4 = i3
```

In this situation, original and copy aren't independent because, despite being a struct, AnyIterator doesn't have value semantics. The box object AnyIterator uses to store its base iterator is a class instance, and when we assigned i3 to i4, only the reference to the box got copied. The storage of the box is shared between the two iterators. Any calls to next on either i3 or i4 now increment the same underlying iterator:

```
i3.next() // Optional(4)
i4.next() // Optional(5)
i3.next() // Optional(6)
i3.next() // Optional(7)
```

Obviously, this could lead to bugs, although in all likelihood, you'll rarely encounter this particular problem in practice. Iterators are usually not something you pass around in your code. You're much more likely to create one locally — sometimes explicitly, but mostly implicitly through a for loop — use it once to loop over the elements, and then throw it away. If you find yourself sharing iterators with other objects, consider wrapping the iterator in a sequence instead.

Function-Based Iterators and Sequences

AnyIterator has a second initializer that takes the next function directly as its argument. Together with the corresponding

AnySequence type, this allows us to create iterators and sequences without defining any new types. For example, we could've defined the Fibonacci iterator alternatively as a function that returns an AnyIterator:

```
func fibsIterator() -> AnyIterator<Int> {
    var state = (0, 1)
    return AnyIterator {
        let upcomingNumber = state.0
        state = (state.1, state.0 + state.1)
        return upcomingNumber
    }
}
```

By keeping the state variable outside of the iterator's next closure and capturing it inside the closure, the closure can mutate the state every time it's invoked. There's only one functional difference between the two Fibonacci iterators: the definition using a custom struct has value semantics, and the definition using AnyIterator doesn't.

Creating a sequence out of this is even easier now because AnySequence provides an initializer that takes a function, which in turn produces an iterator:

```
let fibsSequence = AnySequence(fibsIterator)
Array(fibsSequence.prefix(10)) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Another alternative is to use the sequence function, which has two variants. The first, sequence(first:next:), returns a sequence whose first element is the first argument you passed in; subsequent elements are produced by the closure passed in the next argument. The other variant, sequence(state:next:), is even more powerful because it can keep some arbitrary mutable state around between invocations of the next closure. We can use this to build the Fibonacci sequence with a single function call:

```
let fibsSequence2 = sequence(state: (0, 1)) {
    // The compiler needs a little type inference help here
    (state: inout (Int, Int)) -> Int? in
    let upcomingNumber = state.0
    state = (state.1, state.0 + state.1)
    return upcomingNumber
}
```

```
Array(fibsSequence2.prefix(10)) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

> *The return type of sequence(first:next:) and sequence(state:next:) is UnfoldSequence. This term comes from functional programming, where the same operation is often called unfold. sequence is the natural counterpart to reduce (which is often called fold in functional languages). Where reduce reduces (or folds) a sequence into a single return value, sequence unfolds a single value to generate a sequence.*

The two sequence functions are extremely versatile. They're often a good fit for replacing a traditional C-style for loop that uses non-linear math. In the chapter on structs and classes, we'll talk more about inout parameters.

Infinite Sequences

Like all iterators we've seen so far, the sequence functions apply their next closures lazily, i.e. the next value isn't computed until it's requested by the caller. This makes constructs like fibsSequence2.prefix(10) work. prefix(10) only asks the sequence for its first (up to) 10 elements and then stops. If the sequence had tried to compute all its values eagerly, the program would've crashed with an integer overflow before the next step had a chance to run.

The possibility of creating infinite sequences is one thing that sets sequences apart from collections, which can't be infinite.

Unstable Sequences

Sequences aren't limited to classic collection data structures, such as arrays or lists. Network streams, files on disk, streams of UI events, and many other kinds of data can all be modeled as sequences. But not all of these behave like an array when you iterate over the elements more than once.

While the Fibonacci sequence isn't affected by a traversal of its elements (and a subsequent traversal starts again at zero), a sequence that represents a stream of network packets is consumed by the traversal; it won't produce the same values again if you do another iteration. Both are valid sequences, though, so the documentation is very clear that `Sequence` makes no guarantee about multiple traversals:

> *The `Sequence` protocol makes no requirement on conforming types regarding whether they will be destructively consumed by iteration. As a consequence, don't assume that multiple `for-in` loops on a sequence will either resume iteration or restart from the beginning:*
>
> ```
> for element in sequence {
>     if ... some condition { break }
> }
>
> for element in sequence {
>     // No defined behavior
> }
> ```
>
> *A conforming sequence that is not a collection is allowed to produce an arbitrary sequence of elements in the second `for-in` loop.*

This also explains why the seemingly trivial `first` property is only available on collections, and not on sequences. Invoking a property getter ought to be non-destructive, and only the `Collection` protocol guarantees safe multi-pass iteration.

As an example of a destructively consumed sequence, consider this wrapper on the `readLine` function, which reads lines from the standard input:

```
let standardIn = AnySequence {
    return AnyIterator {
        readLine()
    }
}
```

Now you can use this sequence with the various extensions of Sequence. For example, you could write a line-numbering version of the Unix cat utility:

```
let numberedStdIn = standardIn.enumerated()
for (i, line) in numberedStdIn {
    print("\(i+1):\(line)")
}
```

The enumerated method wraps a sequence in a new sequence that produces pairs of the original sequence's elements and incrementing numbers starting at zero. Just like our wrapper of readLine, elements are lazily generated. The consumption of the base sequence only happens when you move through the enumerated sequence using its iterator, and not when it's created. So if you run the above code from the command line, you'll see it waiting inside the for loop. It prints the lines you type in as you hit return; it does *not* wait until the input is terminated with control-D. But nonetheless, each time enumerated serves up a line from standardIn, it's consuming the standard input. You can't iterate over it twice to get the same results.

As an author of a Sequence extension, you don't need to take into account whether or not the sequence is destructively consumed by iteration. But as a *caller* of a method on a sequence type, you should keep it in mind.

A certain sign that a sequence is stable is if it also conforms to Collection because this protocol makes that guarantee. The reverse isn't true. Even the standard library has some sequences that can be traversed safely multiple times although they aren't collections. Examples include the StrideTo and StrideThrough types, as returned by stride(from:to:by:) and stride(from:through:by:). The fact that you can stride over floating-point numbers would make it tricky (though probably not impossible) to render them as a collection, so they're just sequences.

The Relationship Between Sequences and Iterators

Sequences and iterators are so similar that you may wonder why these need to be separate types at all. Can't we just fold the IteratorProtocol functionality into Sequence? This would indeed work fine for destructively consumed sequences, like our standard input example. Sequences of this kind carry their own iteration state and are mutated as they're traversed.

Stable sequences, like arrays or our Fibonacci sequence, must not be mutated by a for loop; they require separate traversal state, and that's what the iterator provides (along with the traversal logic, but that might as well live in the sequence). The purpose of the makeIterator method is to create this traversal state.

Every iterator can also be seen as an *unstable* sequence over the elements it has yet to return. As a matter of fact, you can turn every iterator into a sequence simply by declaring conformance; Sequence comes with a default implementation for makeIterator that returns self if the conforming type is an iterator. Jordan Rose, a member of the Swift team, said via the swift-evolution mailing list that IteratorProtocol would actually inherit from Sequence if it weren't for language limitations (namely, lack of support for recursive associated type constraints).

Though it may not currently be possible to enforce this relationship, most iterators in the standard library do conform to Sequence.

Subsequences

Sequence has another associated type, named SubSequence:

```
protocol Sequence {
    associatedtype Element
    associatedtype Iterator: IteratorProtocol
        where Iterator.Element == Element
    associatedtype SubSequence
    // ...
}
```

SubSequence is used as the return type for operations that return slices of the original sequence:

- **prefix** and **suffix** — take the first or last *n* elements
- **prefix(while:)** — take elements from the start as long as they conform to a condition
- **dropFirst** and **dropLast** — return subsequences where the first or last *n* elements have been removed
- **drop(while:)** — drop elements until the condition ceases to be true, and then return the rest
- **split** — break up the sequence at the specified separator elements and return an array of subsequences

If you don't specify a type for SubSequence, the compiler will infer it to be AnySequence<Element> because Sequence provides default implementations for the above methods with that return type. If you want to use your own subsequence type, you must provide custom implementations for these methods.

It can sometimes be convenient if SubSequence == Self, i.e. if subsequences have the same type as the base sequence, because this allows you to pass a slice everywhere the base collection is expected. The standard library collections have separate slice types, however; the main motivation for this is to avoid accidental memory "leaks" that can be caused by a tiny slice that keeps its base collection (which may be very large) alive much longer than anticipated. Making slices their own types makes it easier to bind their lifetimes to local scopes.

In an ideal world, the associated type declaration would include constraints to ensure that SubSequence (a) is also a sequence, and (b) has the same element and subsequence types as its base sequence. It should look something like this:

```
associatedtype SubSequence: Sequence
    where Element == SubSequence.Element,
        SubSequence.SubSequence == SubSequence
```

This didn't make it into Swift 4.0 because the compiler lacks support for recursive protocol constraints (Sequence would reference itself). However, we expect these in a future Swift release. Until then, you may find yourself having to add some or all of these constraints to your own Sequence extensions to help the compiler understand the types. The standard library was designed from the start with recursive constraints in mind; many Sequence and Collection APIs will be easier to understand once this feature is implemented because they'll be able to drop the explicit constraints that are currently still necessary.

The following example checks if a sequence starts with the same *n* elements from the head and the tail. It does this by comparing the sequence's prefix with the reversed suffix:

```
extension Sequence where Element: Equatable, SubSequence: Sequence,
    SubSequence.Element == Element
{
    func headMirrorsTail(_ n: Int) -> Bool {
        let head = prefix(n)
        let tail = suffix(n).reversed()
        return head.elementsEqual(tail)
    }
}

[1,2,3,4,2,1].headMirrorsTail(2) // true
```

The comparison using elementsEqual only passes type checking if we tell the compiler that the subsequence is also a sequence, and that its elements have the same type as the base sequence's elements (which we constrained to Equatable).

We show another example of sequence extensions like this in the chapter on generics.

A Linked List

As an example of a custom sequence, let's implement one of the most basic data structures of all: a singly linked list, defined using indirect enums. A linked list node is one of either two things: a node with a value and a reference to the next node, or a node indicating the end of the list. We can define it like this:

```
/// A singly linked list
enum List<Element> {
    case end
    indirect case node(Element, next: List<Element>)
}
```

The indirect keyword here indicates that the compiler should represent the node value as a reference. Swift enums are value types. This means they hold their values directly in the variable, rather than the variable holding a reference to the location of the value. This has many benefits, as we'll see in the structs and classes chapter. But value types can't refer to themselves recursively because doing so would make it impossible for the compiler to calculate their size. The indirect keyword allows an enum case to be held as a reference and thus allows the enum to hold a reference to itself.

We prepend an element to a list by creating a new node, with the next: value set to the current node:

```
let emptyList = List<Int>.end
let oneElementList = List.node(1, next: emptyList)
// node(1, next: List<Swift.Int>.end)
```

We can create a method for prepending to make this process a little easier. We name this method cons, because that's the name of the operation in LISP (it's short for "construct," and adding elements onto the front of the list is sometimes called "consing"):

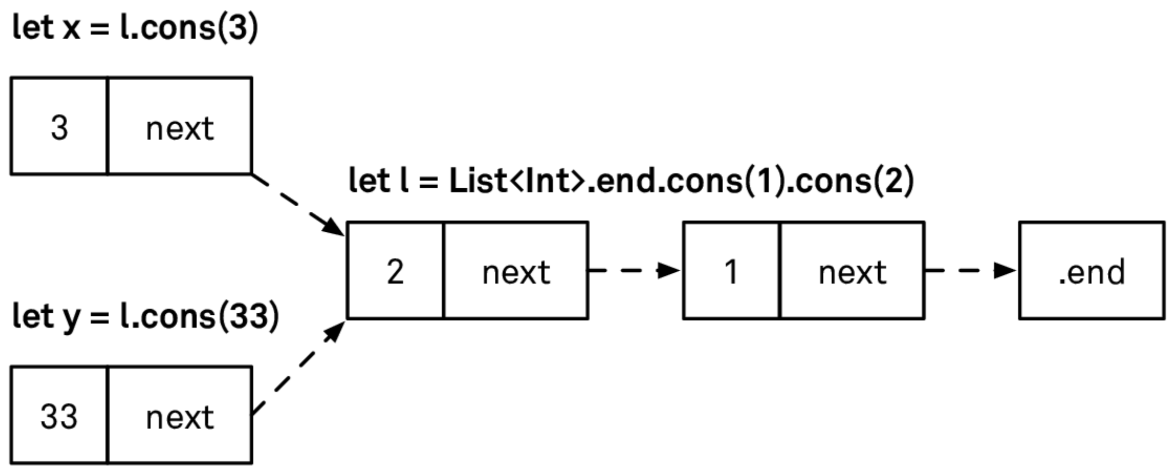
```
extension List {
    /// Return a new list by prepending a node with value x to the
    /// front of a list.
    func cons(_ x: Element) -> List {
        return .node(x, next: self)
    }
}
// A 3-element list, of (3 2 1)
let list = List<Int>.end.cons(1).cons(2).cons(3)
/*
node(3, next: List<Swift.Int>.node(2, next: List<Swift.Int>.node(1,
next: List<Swift.Int>.end)))
*/
```

The chaining syntax makes it clear how a list is constructed, but it's also kind of ugly. We can add conformance to ExpressibleByArrayLiteral to be able to initialize a list with an array literal. The implementation first reverses the input array (because lists are built from the end) and then uses reduce to prepend the elements to the list one by one, starting with the .end node:

```
extension List: ExpressibleByArrayLiteral {
    init(arrayLiteral elements: Element...) {
        self = elements.reversed().reduce(.end) { partialList, element in
            partialList.cons(element)
        }
    }
}
let list2: List = [3,2,1]
/*
node(3, next: List<Swift.Int>.node(2, next: List<Swift.Int>.node(1,
next: List<Swift.Int>.end)))
*/
```

This list type has an interesting property: it's "persistent." The nodes are immutable — once created, you can't change them. Consing another element onto the list doesn't copy the list; it just gives you a new node that links onto the front of the existing list.

This means two lists can share a tail:



List Sharing

The immutability of the list is key here. If you could change the list (say, remove the last entry, or update the element held in a node), then this sharing would be a problem — x might change the list, and the change would affect y.

Still, we can define mutating methods on List to push and pop elements (because the list is also a stack, with consing as push and unwrapping the next element as pop):

```
extension List {
    mutating func push(_ x: Element) {
        self = self.cons(x)
    }

    mutating func pop() -> Element? {
        switch self {
        case .end: return nil
        case let .node(x, next: tail):
            self = tail
            return x
        }
    }
}
```

But didn't we just say that the list had to be immutable for the persistence to work? How can it have mutating methods?

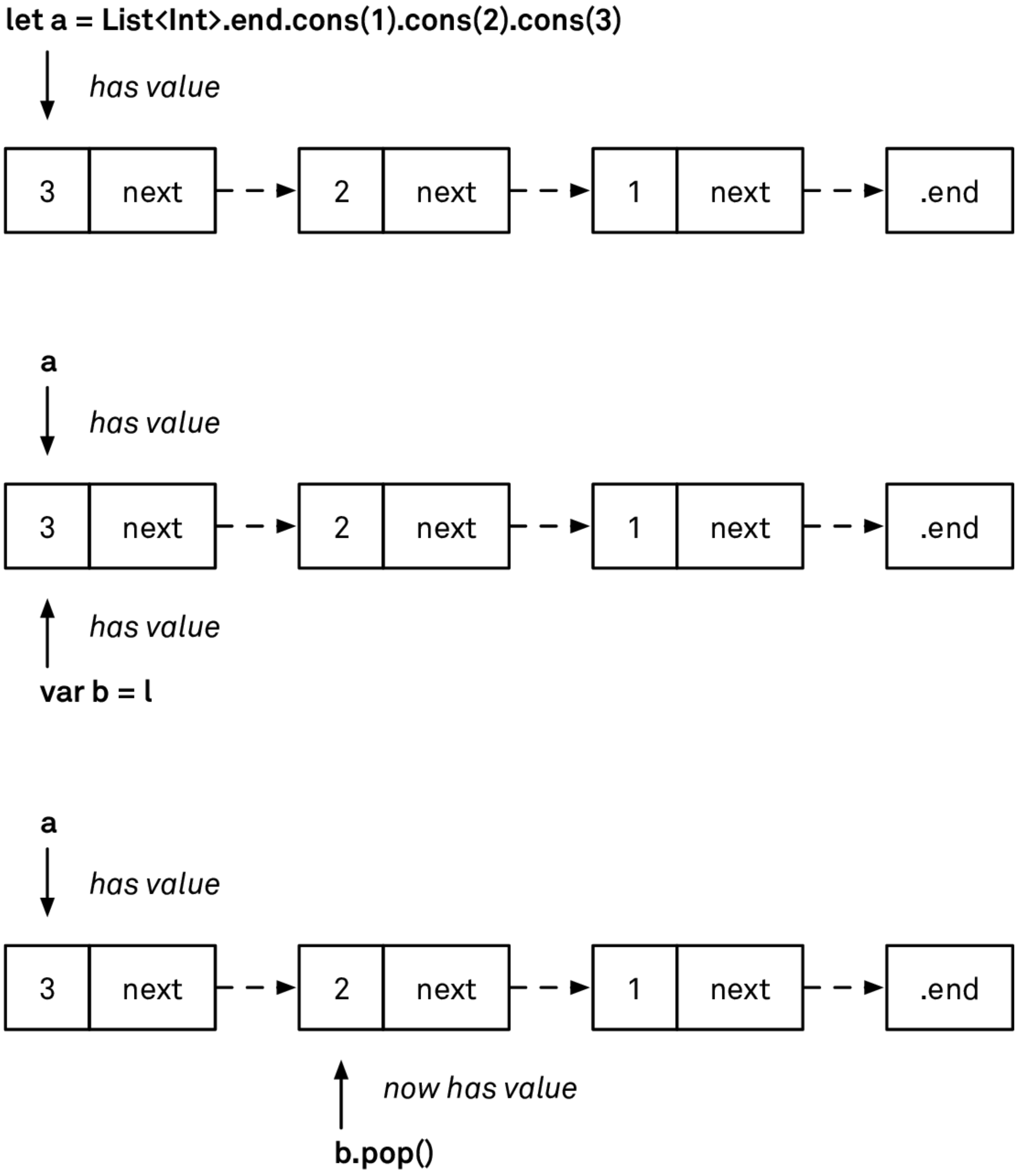
These mutating methods don't change the list. Instead, they just change the part of the list the variables refer to:

```
var stack: List<Int> = [3,2,1]
var a = stack
var b = stack

a.pop() // Optional(3)
a.pop() // Optional(2)
a.pop() // Optional(1)

stack.pop() // Optional(3)
stack.push(4)

b.pop() // Optional(3)
b.pop() // Optional(2)
b.pop() // Optional(1)

stack.pop() // Optional(4)
stack.pop() // Optional(2)
stack.pop() // Optional(1)
```

This shows us the difference between values and variables. The nodes of the list are values; they can't change. A node of three and a reference to the next node can't become some other value; it'll be that value forever, just like the number three can't change. It just is. Just because these values in question are structures with references to each other doesn't make them less value-like.

A variable `a`, on the other hand, can change the value it holds. It can be set to hold a value of an indirect reference to any of the nodes or to the value `end`. But changing `a` doesn't change these nodes; it just changes which node `a` refers to.

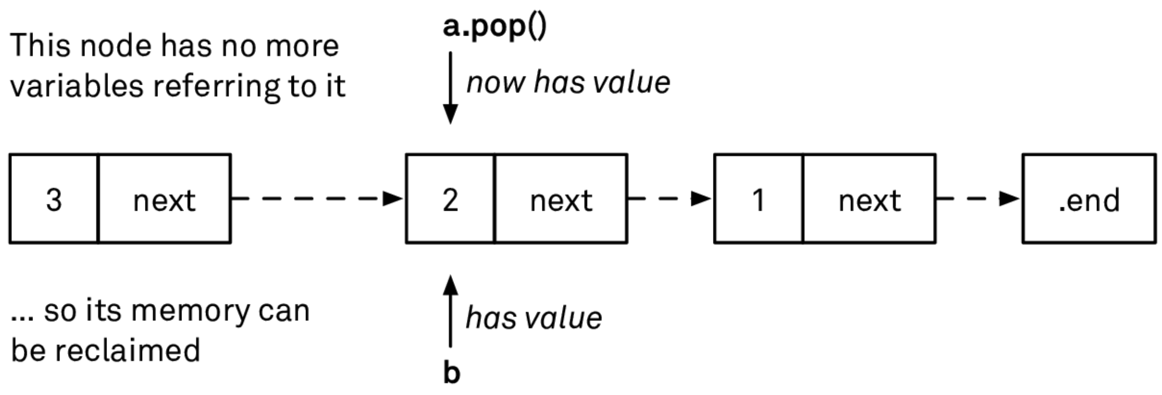
This is what these mutating methods on structs do — they take an implicit `inout` argument of `self`, and they can change the value `self` holds. This doesn't change the list, but rather which part of the list the variable currently represents. In this sense, through `indirect`, the variables have become iterators into the list:



List Iteration

You can, of course, declare your variables with `let` instead of `var`, in which case the variables will be constant (i.e. you can't change the value they hold once they're set). But `let` is about the variables, not the values. Values are constant by definition.

Now this is all just a logical model of how things work. In reality, the nodes are actually places in memory that point to each other. And they take up space, which we want back if it's no longer needed. Swift uses automated reference counting (ARC) to manage this and frees the memory for the nodes that are no longer used:


This node has no more variables referring to it
a.pop()
now has value
3
next
2
next
1
next
.end
... so its memory can be reclaimed
has value
b

List Memory Management

We'll discuss inout in more detail in the chapter on functions, and we'll cover mutating methods as well as ARC in the structs and classes chapter.

Conforming List to Sequence

Since list variables are iterators into the list, this means you can use them to conform List to Sequence. As a matter of fact, List is an example of an unstable sequence that carries its own iteration state, like we saw when we talked about the relationship between sequences and iterators. We can add conformance to IteratorProtocol and Sequence in one go just by providing a next() method; the implementation of this is exactly the same as for pop:

```
extension List: IteratorProtocol, Sequence {
    mutating func next() -> Element? {
        return pop()
    }
}
```

Now you can use lists with for ... in:

```
let list: List = ["1", "2", "3"]
for x in list {
    print("\(x)", terminator: "")
} // 1 2 3
```

This also means that, through the power of protocol extensions, we can use List with dozens of standard library functions:

```
list.joined(separator: ",") // 1,2,3
list.contains("2") // true
list.flatMap { Int($0) } // [1, 2, 3]
list.elementsEqual(["1", "2", "3"]) // true
```

> *In computer science theory, linked lists are more efficient than arrays for some common operations. In practice, however, it's really hard to outperform arrays on modern computer architectures with their extremely fast caches and (relatively) slow main memory. Because arrays use contiguous memory for their elements, the processor can process them way more efficiently. For a good overview of collection performance in Swift, see Károly Lőrentey's book Optimizing Collections.*

Collections

A collection is a stable sequence that can be traversed nondestructively multiple times. In addition to linear traversal, a collection's elements can also be accessed via subscript with an index. Subscript indices are often integers, as they are in arrays. But as we'll see, indices can also be opaque values (as in dictionaries or strings), which sometimes makes working with them non-intuitive. A collection's indices invariably form a finite range, with a defined start and end. This means that unlike sequences, collections can't be infinite.

The Collection protocol builds on top of Sequence. In addition to all the methods inherited from Sequence, collections gain new capabilities that either depend on accessing elements at specific positions or rely on the guarantee of stable iteration, like the count property (if counting the elements of an unstable sequence consumed the sequence, it would kind of defeat the purpose).

Even if you don't need the special features of a collection, you can use Collection conformance to signal to users that your own sequence type is finite and supports multi-pass iteration. It's somewhat strange that you have to come up with an index if all you want is to document that your sequence is multi-pass. Picking a suitable index type to represent positions in the collection is often the hardest part of implementing the Collection protocol. One reason for this design is that the Swift team wanted to avoid the potential confusion of having a distinct protocol for multi-pass sequences that had requirements identical to Sequence but different semantics.

Collections are used extensively throughout the standard library. In addition to Array, Dictionary, and Set, String and its views are all collections, as are CountableRange and UnsafeBufferPointer. Increasingly, we're also seeing types outside the standard library adopt the Collection protocol. Two examples of Foundation types that gained a ton of new capabilities in this way are Data and IndexSet.

A Custom Collection

To demonstrate how collections in Swift work, we'll implement one of our own. Probably the most useful container type not present in the Swift standard library is a queue. Swift arrays are able to easily be used as stacks, with append to push and popLast to pop. However, they're not ideal to use as queues. You could use push combined with remove(at: 0), but removing anything other than the last element of an array is an $O(n)$ operation — because arrays are held in contiguous memory, every element has to shuffle down to fill the gap (unlike popping the last element, which can be done in constant time).

Designing a Protocol for Queues

Before we implement a queue, maybe we should define what we mean by it. A good way to do this is to define a protocol that describes what a queue is. Let's try the following definition:

```
/// A type that can enqueue and dequeue elements.
protocol Queue {
    /// The type of elements held in self.
    associatedtype Element
    /// Enqueue e ≤ ment to self.
    mutating func enqueue(_ newElement: Element)
    /// Dequeue an element from self.
    mutating func dequeue() -> Element?
}
```

As simple as this is, it says a lot about what our definition of queue is: it's defined generically. It can contain any type, represented by the associated type Element. It imposes no restrictions on what Element is.

It's important to note that the comments above the methods are as much a part of a protocol as the actual method names and types. Here, what we don't say tells us as much as what we do: there's no guarantee of the complexity of enqueue or dequeue. We could've said, for example, that both should operate in constant ($O(1)$) time. This would give users adopting this protocol a good idea of the performance characteristics of *any* kind of queue implementing this protocol. But it would rule out data structures, such as priority queues, that might have an $O(log_n)$ enqueueing operation.

It also doesn't offer a peek operation to check without dequeuing, which means it could be used to represent a queue that doesn't have such a feature (such as, say, a queue interface over an operating system or networking call that could only pop, not peek). It doesn't specify whether the two operations are thread-safe. It doesn't specify that the queue is a Collection (though the implementation we're about to write will be).

It doesn't even specify that it's a FIFO queue — it could be a LIFO queue, and we could conform Array to it, with append for enqueue and dequeue implemented via isEmpty/popLast.

Speaking of which, here *is* something the protocol specifies: like Array's popLast, dequeue returns an optional. If the queue is empty, it returns nil. Otherwise, it removes and returns the last element. We don't provide an equivalent for Array.removeLast, which traps if you call it on an empty array.

By making dequeue an optional, the most common operation of repeatedly dequeuing an element until the queue is empty becomes a one-liner, along with the safety of not being able to get it wrong:

```
while let x = queue.dequeue() {
    // Process queue element
}
```

The downside is the inconvenience of always having to unwrap, even when you already know the queue *can't* be empty. The right tradeoff for your particular data type depends on how you envision it to be used. (Conforming your custom collection to the Collection protocol gives you both variants for free, anyway, since Collection provides both a popFirst and a removeFirst method.)

A Queue Implementation

Now that we've defined what a queue is, let's implement it. Below is a very simple FIFO queue, with just enqueue and dequeue methods implemented on top of a couple of arrays.

Since we've named our queue's generic placeholder Element, the same name as the required associated type, there's no need to define it. It's not necessary to name it Element though — the placeholder is just an arbitrary name of your choosing. If it were named Foo, you could either define typealias Element = Foo, or leave Swift to infer it implicitly from the return types of the enqueue and dequeue implementations:

```
/// An efficient variable-size FIFO queue of elements of type E ≤ ment
struct FIFOQueue<Element>: Queue {
    private var left: [Element] = []
    private var right: [Element] = []

    /// Add an element to the back of the queue.
    /// - Complexity: O(1).
    mutating func enqueue(_ newElement: Element) {
        right.append(newElement)
    }

    /// Removes front of the queue.
    /// Returns nil in case of an empty queue.
    /// - Complexity: Amortized O(1).
    mutating func dequeue() -> Element? {
        if left.isEmpty {
            left = right.reversed()
            right.removeAll()
        }
        return left.popLast()
    }
}
```

This implementation uses a technique of simulating a queue through the use of two stacks (two regular arrays). As elements are enqueued, they're pushed onto the "right" stack. Then, when elements are dequeued, they're popped off the "left" stack, where they're held in reverse order. When the left stack is empty, the right stack is reversed onto the left stack.

You might find the claim that the dequeue operation is $O(1)$ slightly surprising. Surely it contains a reverse call that is $O(n)$? But while this is true, the overall *amortized* time to pop an item is constant — over a large number of pushes and pops, the time taken for them all is constant, even though the time for individual pushes or pops might not be.

The key to why this is lies in understanding how often the reverse happens and on how many elements. One technique to analyze this is the "banker's methodology." Imagine that each time you put an element on the queue, you pay a token into the bank. Single enqueue, single token, so constant cost. Then when it comes time to reverse the right-hand stack onto the left-hand one, you have a token in the bank for every element enqueued, and you use those tokens to pay for the reversal. The account never goes into debit, so you never spend more than you paid.

This kind of reasoning is good for explaining why the "amortized" cost of an operation over time is constant, even though individual calls might not be. The same kind of justification can be used to explain why appending an element to an array in Swift is a constant time operation. When the array runs out of storage, it needs to allocate bigger storage and copy all its existing elements into it. But since the storage size doubles with each reallocation, you can use the same "append an element, pay a token, double the array size, spend all the tokens but no more" argument.

Conforming to Collection

We now have a container that can enqueue and dequeue elements. The next step is to add `Collection` conformance to `FIFOQueue`. Unfortunately, figuring out the minimum set of implementations you must provide to conform to a protocol can sometimes be a frustrating exercise in Swift.

At the time of writing, the `Collection` protocol has a whopping six associated types, four properties, seven instance methods, and two subscripts:

```swift
protocol Collection: Sequence {
    associatedtype Element // inherited from Sequence
    associatedtype Index: Comparable

    associatedtype IndexDistance: SignedInteger = Int
    associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>
        where Iterator.Element == Element
    associatedtype SubSequence: Sequence
        /* ... */
    associatedtype Indices: Sequence = DefaultIndices<Self>
        /* ... */

    var first: Element? { get }
    var indices: Indices { get }
    var isEmpty: Bool { get }
    var count: IndexDistance { get }

    func makeIterator() -> Iterator
    func prefix(through: Index) -> SubSequence
    func prefix(upTo: Index) -> SubSequence
    func suffix(from: Index) -> SubSequence
    func distance(from: Index, to: Index) -> IndexDistance
    func index(_: Index, offsetBy: IndexDistance) -> Index
    func index(_: Index, offsetBy: IndexDistance, limitedBy: Index) -> Index?

    subscript(position: Index) -> Element { get }
    subscript(bounds: Range<Index>) -> SubSequence { get }
}
```

(Note that this is an idealized view of the protocol. In Swift 4.0, some of these requirements are really defined on two hidden protocols named _Indexable and _IndexableBase, which Collection extends. This construction is a workaround for the lack of support for recursive associated type constraints in the compiler that was already fixed in the post-4.0 master branch. Swift 4.1 will no longer require this stopgap measure, and the hidden protocols will be merged into Collection.)

The associated SubSequence type, which is inherited from Sequence, has the following additional constraints (because we're inside Collection, the reference to Sequence isn't recursive anymore; ideally, the subsequence of a collection should also be a Collection, but that would require support for recursive constraints):

```
associatedtype SubSequence: Sequence
    where Element == SubSequence.Element,
        SubSequence == SubSequence.SubSequence
```

The Indices type also has a number of constraints. The most important one connects Index with Indices, specifying that the former is the element type of the latter. The other constraints say that Indices is its own subsequence:

```
    associatedtype Indices: Sequence = DefaultIndices<Self>
        where Index == Indices.Element,
            Indices == Indices.SubSequence,
            Indices.Element == Indices.Index,
            Indices.Index == SubSequence.Index
```

With all the requirements above, conforming to Collection seems like a daunting task. Well, it turns out it's actually not that bad. Notice that all associated types except Index and Element have default values, so you don't need to care about those unless your type has special requirements. The same is true for most of the functions, properties, and subscripts: protocol extensions on Collection provide the default implementations. Some of these extensions have associated type constraints that match the protocol's default associated types; for example, Collection only provides a default implementation of the makeIterator method if its Iterator is an IndexingIterator<Self>:

```
extension Collection where Iterator == IndexingIterator<Self> {
    /// Returns an iterator over the elements of the collection.
    func makeIterator() -> IndexingIterator<Self>
}
```

If you decide that your type should have a different iterator type, you'd have to implement this method.

Working out what's required and what's provided through defaults isn't exactly hard, but it's a lot of manual work, and unless you're very careful not to overlook anything, it's easy to end up in an annoying guessing game with the compiler. The most frustrating part of the process may be that the compiler *has* all the information to guide you; the diagnostics just aren't helpful enough yet.

For the time being, your best hope is to find the minimal conformance requirements spelled out in the documentation, as is in fact the case for Collection:

> *… To add Collection conformance to your type, you must declare at least the following requirements:*
>
> - *The startIndex and endIndex properties*
> - *A subscript that provides at least read-only access to your type's elements*
> - *The index(after:) method for advancing an index into your collection*

So in the end, we end up with these requirements:

```
protocol Collection: Sequence {
    /// A type that represents a position in the collection.
    associatedtype Index: Comparable
    /// The position of the first element in a nonempty collection.
    var startIndex: Index { get }
    /// The collection's "past the end" position---that is, the position one
    /// greater than the last valid subscript argument.
    var endIndex: Index { get }
    /// Returns the position immediately after the given index.
    func index(after i: Index) -> Index
    /// Accesses the element at the specified position.
    subscript(position: Index) -> Element { get }
}
```

We can conform FIFOQueue to Collection like so:

```
extension FIFOQueue: Collection {
    public var startIndex: Int { return 0 }
    public var endIndex: Int { return left.count + right.count }

    public func index(after i: Int) -> Int {
        precondition(i < endIndex)
        return i + 1
    }

    public subscript(position: Int) -> Element {
        precondition((0..<endIndex).contains(position), "Index out of bounds")
        if position < left.endIndex {
            return left[left.count - position - 1]
        } else {
            return right[position - left.count]
        }
    }
}
```

We use Int as our queue's Index type. We don't specify an explicit typealias for the associated type; just like with Element, Swift can infer it from the method and property definitions. Note that since the indexing returns elements from the front first, Queue.first returns the next item that will be dequeued (so it serves as a kind of peek).

With just a handful of lines, queues now have more than 40 methods and properties at their disposal. We can iterate over queues:

```
var q = FIFOQueue<String>()
for x in ["1", "2", "foo", "3"] {
    q.enqueue(x)
}

for s in q {
    print(s, terminator: " ")
} // 1 2 foo 3
```

We can pass queues to methods that take sequences:

```
var a = Array(q) // ["1", "2", "foo", "3"]
a.append(contentsOf: q[2...3])
a // ["1", "2", "foo", "3", "foo", "3"]
```

We can call methods and properties that extend Sequence:

```
q.map { $0.uppercased() } // ["1", "2", "FOO", "3"]
q.flatMap { Int($0) } // [1, 2, 3]
q.filter { $0.count > 1 } // ["foo"]
q.sorted() // ["1", "2", "3", "foo"]
q.joined(separator: " ") // 1 2 foo 3
```

And we can call methods and properties that extend Collection:

```
q.isEmpty // false
q.count // 4
q.first // Optional("1")
```

Conforming to ExpressibleByArrayLiteral

When writing a collection like this, it's nice to implement ExpressibleByArrayLiteral too, as we did with our linked list type. This will allow users to create a queue using the familiar [value1, value2, etc] syntax. This can be done easily, like so:

```
extension FIFOQueue: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: Element...) {
        left = elements.reversed()
        right = []
    }
}
```

For our queue logic, we want to reverse the elements to have them ready for use on the left-hand buffer. Of course, we could just copy the elements to the right-hand buffer, but since we're going to be copying elements anyway, it's more efficient to copy them in reverse order so that they don't need reversing later when they're dequeued.

Now queues can easily be created from literals:

```
let queue: FIFOQueue = [1,2,3] // FIFOQueue<Int>(left: [3, 2, 1], right: [])
```

Literals

It's important to underline the difference between literals and types in Swift here. [1,2,3] is *not* an array. Rather, it's an *array literal* — something that can be used to create any type that conforms to ExpressibleByArrayLiteral. This particular literal contains other literals — integer literals — which can create any type that conforms to ExpressibleByIntegerLiteral.

These literals have "default" types — types that Swift will assume if you don't specify an explicit type when you use a literal. So array literals default to Array, integer literals default to Int, float literals default to Double, and string literals default to String. But this only occurs in the absence of you specifying otherwise. For example, the queue declared above is a queue of integers, but it could've been a queue of some other integer type:

```
let byteQueue: FIFOQueue<UInt8> = [1,2,3]
// FIFOQueue<UInt8>(left: [3, 2, 1], right: [])
```

Often, the type of the literal can be inferred from the context. For example, this is what it looks like if a function takes a type that can be created from literals:

```
func takesSetOfFloats(floats: Set<Float>) {
    //...
}

takesSetOfFloats(floats: [1,2,3])
```

This literal will be interpreted as Set<Float>, not as Array<Int>.

Associated Types

We've seen that Collection provides defaults for all but two of its associated types; types adopting the protocol only have to specify an Element and an Index type. While you don't *have* to care much about the other associated types, it's a good idea to take a brief look at each of them in order to better understand what their specific purposes are. Let's go through them one by one.

Iterator. Inherited from Sequence. We already looked at iterators in detail in our discussion on sequences. The default iterator type for collections is IndexingIterator<Self>. This is a simple struct that wraps the collection and uses the collection's own indices to step over each element. The implementation looks like this:

```
public struct IndexingIterator<Elements: _IndexableBase>
    : IteratorProtocol, Sequence
{
    internal let _elements: Elements
    internal var _position: Elements.Index

    init(_elements: Elements) {
        self._elements = _elements
        self._position = _elements.startIndex
    }

    public mutating func next() -> Elements.Element? {
        if _position == _elements.endIndex { return nil }
        let element = _elements[_position]
        _elements.formIndex(after: &_position)
        return element
    }
}
```

(The generic constraint <Elements: _IndexableBase> should really be <Elements: Collection> once the compiler allows recursive associated type constraints.)

Most collections in the standard library use IndexingIterator as their iterator. There should be little reason to write your own iterator type for a custom collection.

SubSequence. Also inherited from Sequence, but Collection restates this type with tighter constraints. A collection's SubSequence should itself also be a Collection. (We say "should" rather than "must" because this requirement is currently not fully expressible in the type system.) The default is Slice<Self>, which wraps the original collection and stores the slice's start and end index in terms of the base collection.

It can make sense for a collection to customize its SubSequence type, especially if it can be Self (i.e. a slice of the collection has the same type as the collection itself). Foundation's Data is an example of a such collection. We'll talk more about slicing later in this chapter.

IndexDistance. A signed integer type that represents the number of steps between two indices. Setting this to anything other than the default Int is rarely necessary. An example where it makes sense to change IndexDistance to (say) Int64 is a collection for accessing files that has to work with very large files (> 4 GB) on 32-bit systems.

Indices. The return type of the collection's indices property. It represents a collection containing all indices that are valid for subscripting the base collection, in ascending order. Note that the endIndex is not included because it signifies the "past the end" position and thus is not a valid subscript argument.

The default type is the imaginatively named DefaultIndices<Self>. Like Slice, it's a simple wrapper for the base collection and a start and end index — it needs to keep a reference to the base collection to be able to advance the indices. This can lead to unexpected performance problems if users mutate the collection while iterating over its indices: if the collection is implemented using copy-on-write (as all collections in the standard library are), the extra reference to the collection can trigger unnecessary copies.

We cover copy-on-write in the chapter on structs and classes. For now, it's enough to know that if your custom collection can provide an alternative Indices type that doesn't need to keep a reference to the base collection, doing so is a worthwhile optimization. This is true for all collections whose index math doesn't rely on the collection itself, like arrays or our queue. If your index is an integer type, you can use CountableRange<Index>:

```
extension FIFOQueue: Collection {
    ...
    typealias Indices = CountableRange<Int>
    var indices: CountableRange<Int> {
        return startIndex..<endIndex
    }
}
```

Indices

An index represents a position in the collection. Every collection has two special indices, startIndex and endIndex. The startIndex designates the collection's first element, and endIndex is the index that comes *after* the last element in the collection. As a result, endIndex isn't a valid index for subscripting; you use it to form ranges of indices (someIndex..<endIndex) or to compare other indices against, e.g. as the break condition in a loop (while someIndex < endIndex).

Up to this point, we've been using integers as the index into our collections. Array does this, and (with a bit of manipulation) our FIFOQueue type does too. Integer indices are very intuitive, but they're not the only option. The only requirement for a collection's Index is that it must be Comparable, which is another way of saying that indices have a defined order.

Take Dictionary, for instance. It would seem that the natural candidates for a dictionary's indices would be its keys; after all, the keys are what we use to address the values in the dictionary. But the key can't be the index because you can't advance it — there's no way to tell what the next index after a given key would be. Also, subscripting with an index is expected to give direct element access, without detours for searching or hashing.

As a result, DictionaryIndex is an opaque value that points to a position in the dictionary's internal storage buffer. It really is just a wrapper for a single Int offset, but that's an implementation detail of no interest to users of the collection. (In fact, the reality is somewhat more complex because dictionaries that get passed to or returned from Objective-C APIs use an NSDictionary as their backing store for efficient bridging, and the index type for those dictionaries is different. But you get the idea.)

This also explains why subscripting a Dictionary with an index doesn't return an optional value, whereas subscripting with a key does. The subscript(_ key: Key) we're so used to is an additional overload of the subscript operator that's defined directly on Dictionary. It returns an optional Value:

```
struct Dictionary {
    ...
    subscript(key: Key) -> Value?
}
```

In contrast, subscripting with an index is part of the Collection protocol and *always* returns a non-optional value, because addressing a collection with an invalid index (like an out-of-bounds index on an array) is considered a programmer error and doing so is supposed to trap:

```
protocol Collection {
    subscript(position: Index) -> Element { get }
}
```

Notice the return type Element. A dictionary's Element type is the tuple type (key: Key, value: Value), so for Dictionary, this subscript returns a key-value pair and not just a Value. This is also why iterating over a dictionary in a for loop produces key-value pairs.

In the section on array indexing in the built-in collections chapter, we discussed why it makes sense even for a “safe” language like Swift not to wrap every failable operation in an optional or error construct. “If every API can fail, then you can’t write useful code. You need to have some fundamental basis that you can rely on, and trust to operate correctly,” otherwise your code gets bogged down in safety checks.

Index Invalidation

Indices may become invalid when the collection is mutated. Invalidation could mean that the index remains valid but now addresses a different element or that the index is no longer a valid index for the collection and using it to access the collection will trap. This should be intuitively clear when you consider arrays. When you append an element, all existing indices remain valid. When you remove the first element, an existing index to the last element becomes invalid. Meanwhile smaller indices remain valid, but the elements they point to have changed.

A dictionary index remains stable when new key-value pairs are added *until* the dictionary grows so much that it triggers a reallocation. This is because the element’s location in the dictionary’s storage buffer doesn’t change, as elements are inserted until the buffer has to be resized, forcing all elements to be rehashed. Removing elements from a dictionary invalidates indices.

An index should be a dumb value that only stores the minimal amount of information required to describe an element’s position. In particular, indices shouldn’t keep a reference to their collection, if at all possible. Similarly, a collection usually can’t distinguish one of its “own” indices from one that came from another collection of the same type. Again, this is trivially evident for arrays. Of course, you can use an integer index that was derived from one array to index another:

```
let numbers = [1,2,3,4]
let squares = numbers.map { $0 * $0 }
let numbersIndex = numbers.index(of: 4)! // 3
squares[numbersIndex] // 16
```

This also works with opaque index types, such as String.Index. In this example, we use one string's startIndex to access the first character of another string:

```
let hello = "Hello"
let world = "World"
let helloIdx = hello.startIndex
world[helloIdx] // W
```

However, the fact that you can do this doesn't mean it's generally a good idea. If we had used the index to subscript into an empty string, the program would've crashed because the index was out of bounds.

There are legitimate use cases for sharing indices between collections, though. The biggest one is working with slices. The Collection protocol requires that an index of the base collection must address the same element in a slice of that collection, so it's always safe to share indices with slices.

Advancing Indices

Swift 3 introduced a major change to the way index traversal is handled for collections. The task of advancing an index forward or backward (i.e. deriving a new index from a given index) is now a responsibility of the collection, whereas up until Swift 2, indices were able to advance themselves. Where you used to write someIndex.successor() to step to the next index, you now write collection.index(after: someIndex).

Why did the Swift team decide to make this change? In short, performance. It turns out that deriving an index from another very often requires information about the collection's internals. It doesn't for arrays, where advancing an index is a simple addition operation. But a string index, for example, needs to inspect the actual character data because characters have variable sizes in Swift.

In the old model of self-advancing indices, this meant that the index had to store a reference to the collection's storage. That extra reference was enough to defeat the copy-on-write optimizations used by the standard library collections and would result in unnecessary copies when a collection was mutated during iteration.

By allowing indices to remain dumb values, the new model doesn't have this problem. It's also conceptually easier to understand and can make implementations of custom index types simpler. When you do implement your own index type, keep in mind that the index shouldn't keep a reference to the collection if at all possible. In most cases, an index can likely be represented with one or two integers that efficiently encode the position to the element in the collection's underlying storage.

The downside of the new indexing model is a more verbose syntax.

Custom Collection Indices

As an example of a collection with non-integer indices, we'll build a way to iterate over the words in a string. When you want to split a string into its words, the easiest way is to use `split(separator:maxSplits:omittingEmptySubsequences:)`. This method is defined on `Collection`, and it turns a collection into an array of `SubSequence`s, split at the provided separator:

```
var str = "Still I see monsters"
str.split(separator: " ") // ["Still", "I", "see", "monsters"]
```

This code returns an array of words. Each word is of type `Substring`, which is `String`'s associated `SubSequence` type. Whenever you need to split a collection, the split method is almost always the right tool to use. It does have one disadvantage though: it eagerly computes the entire array. If you have a large string and only need the first few words, this isn't very efficient. To be more efficient, we build a `Words` collection that doesn't compute all the words up front but instead allows us to iterate lazily.

Let's start by finding the range of the first word in a Substring. We'll use spaces as word boundaries, although it's an interesting exercise to make this configurable. A substring might start with an arbitrary number of spaces, which we skip over. start is the substring with the leading spaces removed. We'll then try to find the next space; if there's any space, we use that as the end of the word boundary. If we can't find any more spaces, we use endIndex:

```
extension Substring {
    var nextWordRange: Range<Index> {
        let start = drop(while: { $0 == " "})
        let end = start.index(where: { $0 == " "}) ?? endIndex
        return start.startIndex..<end
    }
}
```

Note that Range is half-open: the upper bound end isn't included in the word range.

A logical first choice for the index type of a Words collection would be Int: the index i would mean the ith word in the collection. However, accessing an element through the index-based subscript needs to be $O(1)$, and in order to find the i$^{\text{th}}$ word, we'd have to process the entire string (which is an $O(n)$ operation).

Another choice for the index type would be to use String.Index. The collection's startIndex would be string.startIndex, the index after that would be the index of the beginning of the next word, and so on. Unfortunately, the subscript implementation will have a similar problem: finding the end of the word is also $O(n)$.

Instead, we make our index a wrapper around Range<Substring.Index>. A collection's index needs to be Comparable (and because Comparable inherits from Equatable, we need to implement == as well). To compare two indices, we only compare the range's lower bounds. This doesn't work for comparing Ranges in general, but for our purpose it suffices. By marking the range property and the initializer as fileprivate, we make WordsIndex an opaque type; users of our collection don't know the internal structure, and the only way to create an index is through the collection's interface:

```
struct WordsIndex: Comparable {
    fileprivate let range: Range<Substring.Index>
    fileprivate init(_ value: Range<Substring.Index>) {
        self.range = value
    }

    static func <(lhs: Words.Index, rhs: Words.Index) -> Bool {
        return lhs.range.lowerBound < rhs.range.lowerBound
    }

    static func ==(lhs: Words.Index, rhs: Words.Index) -> Bool {
        return lhs.range == rhs.range
    }
}
```

We're now ready to build our Words collection. It stores the underlying String as a Substring (we'll see why in the section on slicing) and provides a start index and an end index. The Collection protocol requires startIndex to have a complexity of $O(1)$. Unfortunately, computing it takes $O(n)$, where n is the number of spaces at the start of the string. Therefore, we compute it in the initializer and store it, instead of defining it as a computed property. For the end index, we use an empty range that's outside the bounds of the underlying string:

```
struct Words: Collection {
    let string: Substring
    let startIndex: WordsIndex

    init(_ s: String) {
        self.init(s[...])
    }

    private init(_ s: Substring) {
        self.string = s
        self.startIndex = WordsIndex(string.nextWordRange)
    }

    var endIndex: WordsIndex {
        let e = string.endIndex
        return WordsIndex(e..<e)
    }
}
```

Collection also requires us to provide a subscript that accesses elements. Here we can directly use the index's underlying range. Note that using the range of the word as the index makes the implementation $O(1)$:

```
extension Words {
    subscript(index: WordsIndex) -> Substring {
        return string[index.range]
    }
}
```

As the final Collection requirement, we need a way to compute the index following a given index. The upper bound of the index's range is *not* included, so unless that's already the string's endIndex, we can take the substring from the upper bound onward, and then look for the next word range:

```
extension Words {
    func index(after i: WordsIndex) -> WordsIndex {
        guard i.range.upperBound < string.endIndex
            else { return endIndex }
        let remainder = string[i.range.upperBound...]
        return WordsIndex(remainder.nextWordRange)
    }
}

Array(Words(" hello world test ").prefix(2)) // ["hello", "world"]
```

With some effort, the Words collection could be changed to solve more general problems. First of all, we can make the word boundary configurable: instead of using a space, we can pass in a function, isWordBoundary: (Character) -> Bool. Second, the code isn't really specific to strings: we could replace String with any kind of collection. For example, we could reuse the same algorithm to lazily split Data into processable chunks.

Slices

All collections get a default implementation of the slicing operation and have an overload for subscript that takes a Range<Index>. This is the equivalent of list.dropFirst():

```
let words: Words = Words("one two three")
let onePastStart = words.index(after: words.startIndex)
let firstDropped = words[onePastStart..<words.endIndex]
Array(firstDropped) // ["two", "three"]
```

Since operations like words[somewhere..<words.endIndex] (slice from a specific point to the end) and words[words.startIndex..<somewhere] (slice from the start to a specific point) are common, there are variants in the standard library that do these operations in a more readable way:

```
let firstDropped2 = words.suffix(from: onePastStart)
// or:
let firstDropped3 = words[onePastStart...]
```

By default, the type of firstDropped won't be Words — it'll be a Slice<Words>. Slice is a lightweight wrapper on top of any collection. The implementation looks something like this:

```
struct Slice<Base: Collection>: Collection {
    typealias Index = Base.Index
    typealias IndexDistance = Base.IndexDistance
    typealias SubSequence = Slice<Base>

    let collection: Base

    var startIndex: Index
    var endIndex: Index

    init(base: Base, bounds: Range<Index>) {
        collection = base
        startIndex = bounds.lowerBound
        endIndex = bounds.upperBound
    }

    func index(after i: Index) -> Index {
        return collection.index(after: i)
    }

    subscript(position: Index) -> Base.Element {
        return collection[position]
    }

    subscript(bounds: Range<Base.Index>) -> Slice<Base> {
        return Slice(base: collection, bounds: bounds)
    }
}
```

Slice is a perfectly good default subsequence type, but every time you write a custom collection, it's worth investigating whether or not you can make the collection its own SubSequence. For Words, this is easy to do:

```
extension Words {
    subscript(range: Range<WordsIndex>) -> Words {
        let start = range.lowerBound.range.lowerBound
        let end = range.upperBound.range.upperBound
        return Words(string[start..<end])
    }
}
```

The compiler infers the SubSequence type from the return type of the range-based subscript.

Using the same type for a collection and its SubSequence makes life easier for users of the collection because they only have to understand a single type instead of two. On the flip side, using distinct types for "root" collections and their slices can help prevent accidental memory "leaks," which is why the standard library has ArraySlice and Substring.

Slices Share Indices with the Base Collection

A formal requirement of the Collection protocol is that indices of a slice can be used interchangeably with indices of the original collection:

> *A collection and its slices share the same indices. An element of a collection is located under the same index in a slice as in the base collection, so long as neither the collection nor the slice has been mutated since the slice was created.*

An important implication of this model is that, even when using integer indices, a collection's index won't necessarily start at zero. Here's an example of the start and end indices of an array slice:

```
let cities = ["New York", "Rio", "London", "Berlin",
    "Rome", "Beijing", "Tokyo", "Sydney"]
let slice = cities[2...4]
cities.startIndex // 0
cities.endIndex // 8
slice.startIndex // 2
slice.endIndex // 5
```

Accidentally accessing slice[0] in this situation will crash your program. This is another reason to always prefer constructs like for x in collection or for index in collection.indices over manual index math if possible — with one exception: if you mutate a collection while iterating over its indices, any strong reference the indices object holds to the original collection will defeat the copy-on-write optimizations and may cause an unwanted copy to be made. Depending on the size of the collection, this can have a significant negative performance impact. (Not all collections use an Indices type that strongly references the base collection, but many do because that's what the standard library's DefaultIndices type does.)

To avoid this issue, you can replace the for loop with a while loop and advance the index manually in each iteration, thus avoiding the indices property. Just remember that if you do this, always start the loop at collection.startIndex and not at 0.

A Generic PrefixIterator

Now that we know any collection can be sliced, we could revisit our prefix iterator code from earlier in the chapter and write a version that works with any collection:

```
struct PrefixIterator<Base: Collection>: IteratorProtocol, Sequence {
    let base: Base
    var offset: Base.Index

    init(_ base: Base) {
        self.base = base
        self.offset = base.startIndex
    }

    mutating func next() -> Base.SubSequence? {
        guard offset != base.endIndex else { return nil }
        base.formIndex(after: &offset)
        return base.prefix(upTo: offset)
    }
}
```

By conforming the iterator directly to Sequence, we can use it with functions that work on sequences without having to define another type:

```
let numbers = [1,2,3]
Array(PrefixIterator(numbers))
// [ArraySlice([1]), ArraySlice([1, 2]), ArraySlice([1, 2, 3])]
```

Specialized Collections

Like all well-designed protocols, Collection strives to keep its requirements as small as possible. In order to allow a wide array of types to become collections, the protocol shouldn't require conforming types to provide more than is absolutely necessary to implement the desired functionality.

Two particularly interesting limitations are that a Collection can't move its indices backward, and that it doesn't provide any functionality for mutating the collection, such as inserting, removing, or replacing elements. That isn't to say that conforming types can't have these capabilities, of course, only that the protocol makes no assumptions about them.

Some algorithms have additional requirements, though, and it would be nice to have generic variants of them, even if only some collections can use them. For this purpose, the standard library includes four specialized collection protocols, each of which refines Collection in a particular way in order to enable new functionality (the quotes are from the standard library documentation):

- **BidirectionalCollection** — "A collection that supports backward as well as forward traversal."

- **RandomAccessCollection** — "A collection that supports efficient random-access index traversal."
- **MutableCollection** — "A collection that supports subscript assignment."
- **RangeReplaceableCollection** — "A collection that supports replacement of an arbitrary subrange of elements with the elements of another collection."

Let's discuss them one by one.

BidirectionalCollection

BidirectionalCollection adds a single but critical capability: the ability to move an index backward using the index(before:) method. This is sufficient to give your collection a default last property, matching first:

```
extension BidirectionalCollection {
    /// The last element of the collection.
    public var last: Element? {
        return isEmpty ? nil : self[index(before: endIndex)]
    }
}
```

Collection could certainly provide a last property itself, but that wouldn't be a good idea. To get the last element of a forward-only collection, you have to iterate all the way to the end, i.e. an $O(n)$ operation. It'd be misleading to provide a cute little property for the last element — a singly linked list with a million elements takes a long time to fetch the last element.

An example of a bidirectional collection in the standard library is String. For Unicode-related reasons that we'll go into in the chapter on strings, a character collection can't provide random access to its characters, but you can move backward from the end, character by character.

BidirectionalCollection also adds more efficient implementations of some operations that profit from traversing the collection backward, such as suffix, removeLast, and reversed. The latter doesn't immediately reverse the collection, but instead returns a lazy view:

```
extension BidirectionalCollection {
    /// Returns a view presenting the elements of the collection in reverse
    /// order.
    /// - Complexity: O(1)
    public func reversed() -> ReversedCollection<Self> {
        return ReversedCollection(_base: self)
    }
}
```

Just as with the enumerated wrapper on Sequence, no actual reversing takes place. Instead, ReversedCollection holds the base collection and uses a reversed index into the collection. The collection then reverses the logic of all index traversal methods so that moving forward moves backward in the base collection, and vice versa.

Value semantics play a big part in the validity of this approach. On construction, the wrapper "copies" the base collection into its own storage so that a subsequent mutation of the original collection won't change the copy held by ReversedCollection. This means that it has the same observable behavior as the version of reversed that returns an array. We'll see in the chapter on structs and classes that, in the case of copy-on-write types such as Array (or immutable persistent structures like List, or types composed of two copy-on-write types like FIFOQueue), this is still an efficient operation.

Most types in the standard library that conform to Collection also conform to BidirectionalCollection. However, types like Dictionary and Set don't — mostly as the idea of forward and backward iteration makes little sense for inherently unordered collections.

RandomAccessCollection

A RandomAccessCollection provides the most efficient element access of all — it can jump to any index in constant time. To do this, conforming types must be able to (a) move an index any distance, and (b) measure the distance between any two indices, both in $O(1)$ time. RandomAccessCollection redeclares its associated Indices and SubSequence types with stricter constraints — both must be random-access themselves — but otherwise adds no new requirements over BidirectionalCollection. Adopters must ensure to meet the documented $O(1)$ complexity requirements, however. You can do this either by providing implementations of the index(_:offsetBy:) and distance(from:to:) methods, or by using an Index type that conforms to Strideable, such as Int.

At first, it might seem like RandomAccessCollection doesn't add much. Even a simple forward-traverse-only collection like our Words can advance an index by an arbitrary distance. But there's a big difference. For Collection and BidirectionalCollection, index(_:offsetBy:) operates by incrementing the index successively until it reaches the destination. This clearly takes linear time — the longer the distance traveled, the longer it'll take to run. Random-access collections, on the other hand, can just move straight to the destination.

This ability is key in a number of algorithms, a couple of which we'll look at in the chapter on generics. There, we'll implement a generic binary search, but it's crucial this algorithm be constrained to random-access collections only — otherwise it'd be far less efficient than just searching through the collection from start to end.

A random-access collection can compute the distance between its startIndex and endIndex in constant time, which means the collection can also compute count in constant time out of the box.

MutableCollection

A mutable collection supports in-place element mutation. The single new requirement it adds to Collection is that the single-element subscript now must also have a setter. We can add conformance for our queue type:

```
extension FIFOQueue: MutableCollection {
    public var startIndex: Int { return 0 }
    public var endIndex: Int { return left.count + right.count }

    public func index(after i: Int) -> Int {
        return i + 1
    }

    public subscript(position: Int) -> Element {
        get {
            precondition((0..<endIndex).contains(position), "Index out of bounds")
            if position < left.endIndex {
                return left[left.count - position - 1]
            } else {
                return right[position - left.count]
            }
        }
        set {
            precondition((0..<endIndex).contains(position), "Index out of bounds")
            if position < left.endIndex {
                left[left.count - position - 1] = newValue
            } else {
                return right[position - left.count] = newValue
            }
        }
    }
}
```

Notice that the compiler won't let us add the subscript setter in an extension to an existing Collection; it's neither allowed to provide a setter without a getter, nor can we redefine the existing getter, so we have to replace the existing Collection-conforming extension. Now the queue is mutable via subscripts:

```
var playlist: FIFOQueue = ["Shake It Off", "Blank Space", "Style"]
playlist.first // Optional("Shake It Off")
playlist[0] = "You Belong With Me"
playlist.first // Optional("You Belong With Me")
```

Relatively few types in the standard library adopt MutableCollection. Of the three major collection types, only Array does. MutableCollection allows changing the values of a collection's elements but not the length of the collection or the order of the elements. This last point explains why Dictionary and Set do *not* conform to MutableCollection, although they're certainly mutable data structures.

Dictionaries and sets are *unordered* collections — the order of the elements is undefined as far as the code using the collection is concerned. However, *internally* , even these collections have a stable element order that's defined by their implementation. When you mutate a MutableCollection via subscript assignment, the index of the mutated element must remain stable, i.e. the position of the index in the indices collection must not change. Dictionary and Set can't satisfy this requirement because their indices point to the bucket in their internal storage where the corresponding element is stored, and that bucket could change when the element is mutated.

RangeReplaceableCollection

For operations that require adding or removing elements, use the RangeReplaceableCollection protocol. This protocol requires two things:

- An empty initializer — this is useful in generic functions, as it allows a function to create new empty collections of the same type.
- A replaceSubrange(_:with:) method — this takes a range to replace and a collection to replace it with.

RangeReplaceableCollection is a great example of the power of protocol extensions. You implement one uber-flexible method, replaceSubrange, and from that comes a whole bunch of derived methods for free:

- **append(_:)** and **append(contentsOf:)** — replace endIndex..<endIndex (i.e. replace the empty range at the end) with the new element/elements

- **remove(at:)** and **removeSubrange(_:)** — replace i...i or subrange with an empty collection

- **insert(at:)** and **in sert(contentsOf:at:)** — replace i..<i (i.e. replace the empty range at that point in the array) with a new element/elements

- **removeAll** — replace startIndex..<endIndex with an empty collection

If a specific collection type can use knowledge about its implementation to perform these functions more efficiently, it can provide custom versions that will take priority over the default protocol extension ones.

We chose to have a very simple inefficient implementation for our queue type. As we stated when defining the data type, the left stack holds the element in reverse order. In order to have a simple implementation, we need to reverse all the elements and combine them into the right array so that we can replace the entire range at once:

```
extension FIFOQueue: RangeReplaceableCollection {
    mutating func replaceSubrange<C: Collection>(_ subrange: Range<Int>,
        with newElements: C) where C.Element == Element
    {
        right = left.reversed() + right
        left.removeAll()
        right.replaceSubrange(subrange, with: newElements)
    }
}
```

You might like to try implementing a more efficient version, which looks at whether or not the replaced range spans the divide between the left and right stacks. There's no need for us to implement the empty init in this example, since the FIFOQueue struct already has one by default.

Unlike BidirectionalCollection and RandomAccessCollection, where the latter extends the former, RangeReplaceableCollection doesn't inherit from MutableCollection; they form distinct hierarchies. An example of a standard library collection that does conform to RangeReplaceableCollection but isn't a MutableCollection is String. The reasons boil down to what we said above about indices having to remain stable during a single-element subscript mutation, which String can't guarantee. We'll talk more about this in the chapter on strings.

> *The standard library knows twelve distinct kinds of collections that are the result of the combination of three traversal methods (forward-only, bidirectional, and random-access) with four mutability types (immutable, mutable, range-replaceable, and mutable-and-range-replaceable).*
>
> *Because each of these needs a specialized default subsequence type, you may encounter types like* MutableRangeReplaceableBidirectionalSlice. *Don't let these monstrosities discourage you from working with them — they behave just like a normal* Slice, *with extra capabilities that match their base collection, and you rarely need to care about the specific type. And if and when Swift gets conditional protocol conformance, the types will be removed in favor of constrained extensions on* Slice.

Composing Capabilities

The specialized collection protocols can be composed very elegantly into a set of constraints that exactly matches the requirements of a particular algorithm. As an example, take the sort method in the standard library for sorting a collection in place (unlike its non-mutating sibling, sorted, which returns the sorted elements in an array). Sorting in place requires the collection to be mutable. If you want the sort to be fast, you also need random access. Last but not least, you need to be able to compare the collection's elements to each other.

Combining these requirements, the sort method is defined in an extension to MutableCollection, with RandomAccessCollection and Element: Comparable as additional constraints:

```
extension MutableCollection
    where Self: RandomAccessCollection, Element: Comparable {
    /// Sorts the collection in place.
    public mutating func sort() { ... }
}
```

Recap

The Sequence and Collection protocols form the foundation of Swift's collection types. They provide dozens of common operations to conforming types and act as constraints for your own generic functions. The specialized collection types, such as MutableCollection or RandomAccessCollection, give you very fine-grained control over the functionality and performance requirements of your algorithms.

The high level of abstraction necessarily makes the model complex, so don't feel discouraged if not everything makes sense immediately. It takes practice to become comfortable with the strict type system, especially since, more often than not, discerning what the compiler wants to tell you is an art form that forces you to carefully read between the lines. The reward is an extremely flexible system that can handle everything from a pointer to a memory buffer to a destructively consumed stream of network packets.

This flexibility means that once you've internalized the model, chances are that a lot of code you'll come across in the future will instantly feel familiar because it's built on the same abstractions and supports the same operations. And whenever you create a custom type that fits in the Sequence or Collection framework, consider adding the conformance. It'll make life easier both for you and for other developers who work with your code.

The next chapter is all about another fundamental concept in Swift: optionals.

Optionals

Sentinel Values

An extremely common pattern in programming is to have an operation that may or may not return a value.

Perhaps not returning a value is an expected outcome when you've reached the end of a file you were reading, as in the following C snippet:

```
int ch;
while ((ch = getchar()) != EOF) {
    printf("Read character %c\n", ch);
}
printf("Reached end-of-file\n");
```

`EOF` is just a `#define` for `-1`. As long as there are more characters in the file, `getchar` returns them. But if the end of the file is reached, `getchar` returns `-1`.

Or perhaps returning no value means "not found," as in this bit of C++:

```
auto vec = {1, 2, 3};
auto iterator = std::find(vec.begin(), vec.end(), someValue);
if (iterator != vec.end()) {
    std::cout << "vec contains " << *iterator << std::endl;
}
```

Here, `vec.end()` is the iterator "one past the end" of the container; it's a special iterator you can check against the container's end but that you mustn’t ever actually use to access a value — similar to a collection’s endIndex in Swift. The find function uses it to indicate that no such value is present in the container.

Or maybe the value can’t be returned because something went wrong during the function’s processing. Probably the most notorious example is that of the null pointer. This innocuous-looking piece of Java code will likely throw a NullPointerException:

```
int i = Integer.getInteger("123")
```

It happens that Integer.getInteger doesn’t parse strings into integers, but rather gets the integer value of a system property named “123.” This property probably doesn’t exist, in which case getInteger returns null. When the null then gets unboxed into an int, Java throws an exception.

Or take this example in Objective-C:

```
[[NSString alloc] initWithContentsOfURL:url
    encoding:NSUTF8StringEncoding error:&e];
```

Here, the NSString might be nil, in which case — and only then — the error pointer should be checked. There’s no guarantee the error pointer is valid if the result is non-nil.

In all of the above examples, the function returns a special “magic” value to indicate that it hasn’t returned a real value. Magic values like these are called “sentinel values.”

But this approach is problematic. The result returned looks and feels like a real value. An int of -1 is still a valid integer, but you don’t ever want to print it out. v.end() is an iterator, but the results are undefined if you try to use it. And everyone loves seeing a stack dump when your Java program throws a NullPointerException.

Unlike Java, Objective-C allows sending messages to nil. This is "safe" in so far as the runtime guarantees that the return value from a message to nil will always be the equivalent of zero, i.e. nil for object return types, 0 for numeric types, and so on. If the message returns a struct, it'll have all its members initialized to zero. With this in mind, consider the following snippet:

```
NSString *someString = ...;
if ([someString rangeOfString:@"Swift"].location != NSNotFound) {
    NSLog(@"Someone mentioned Swift!");
}
```

If someString is nil, whether accidentally or on purpose, the rangeOfString: message will return a zeroed NSRange. Hence, its .location will be zero, and the inequality comparison against NSNotFound (which is defined as NSIntegerMax) will succeed. Therefore, the body of the if statement will be executed when it shouldn't be.

Null references cause so many problems that Tony Hoare, credited with their creation in 1965, calls them his "billion-dollar mistake":

> *At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

We've seen that sentinel values are error-prone — you can forget to check the sentinel value and accidentally use it. They also require prior knowledge. Sometimes there's an idiom that's widely followed in a community, as with the C++ end iterator or the error handling conventions in Objective-C. If such rules don't exist or you're not aware of them, you have to refer to the documentation. Moreover, there's no way for the function to indicate it *can't* fail. If a call returns a pointer, that pointer might never be nil. But there's no way to tell except by reading the documentation, and even then, the documentation might be wrong.

Solving the Magic Value Problem with Enumerations

Of course, every good programmer knows magic numbers are bad. Most languages support some kind of enumeration type, which is a safer way of representing a set of discrete possible values for a type.

Swift takes enumerations further with the concept of "associated values." These are enumeration values that can also have another value associated with them:

```
enum Optional<Wrapped> {
    case none
    case some(Wrapped)
}
```

In some languages, these are called "tagged unions" (or "discriminated unions") — a union being multiple different possible types all held in the same space in memory, with a tag to tell which type is actually held. In Swift enums, this tag is the enum case.

The only way to retrieve an associated value is through pattern matching in a switch or an if case let statement. Unlike with a sentinel value, you can't accidentally use the value embedded in an Optional without explicitly checking and unpacking it.

So instead of returning an index, the Swift equivalent of find — called index(of:) — returns an Optional<Index> with a protocol extension implementation somewhat similar to this:

```
extension Collection where Element: Equatable {
    func index(of element: Element) -> Optional<Index> {
        var idx = startIndex
        while idx != endIndex {
            if self[idx] == element {
                return .some(idx)
            }
            formIndex(after: &idx)
        }
        // Not found, return .none
        return .none
    }
}
```

Since optionals are fundamental to Swift, there's lots of syntax support to tidy this up: Optional<Index> can be written Index?; optionals conform to ExpressibleByNilLiteral so that you can write nil instead of .none; and non-optional values (like idx) are automatically "upgraded" to optionals where needed so that you can write return idx instead of return .some(idx). The syntactic sugar is effective in disguising the true nature of the Optional type. It's worth remembering that there's nothing magical about it; it's just a normal enum, and if it didn't exist, you could define it yourself.

Now there's no way a user could mistakenly use the value before checking if it's valid:

```
var array = ["one", "two", "three"]
let idx = array.index(of: "four")
// Compile-time error: remove(at:) takes an Int, not an Optional<Int>
array.remove(at: idx)
```

Instead, you're forced to "unwrap" the optional in order to get at the index within, assuming you didn't get none back:

```
var array = ["one", "two", "three"]
switch array.index(of: "four") {
case .some(let idx):
    array.remove(at: idx)
case .none:
    break // Do nothing
}
```

This switch statement writes the enumeration syntax for optionals out longhand, including unpacking the associated value in the some case. This is great for safety, but it's not very pleasant to read or write. A more succinct alternative is to use the ? pattern suffix syntax to match a some optional value, and you can use the nil literal to match none:

```
switch array.index(of: "four") {
case let idx?: // Equivalent to .some(let idx)
    array.remove(at: idx)
case nil:
    break // Do nothing
}
```

But this is still clunky. Let's take a look at all the other ways you can make your optional processing short and clear, depending on your use case.

A Tour of Optional Techniques

Optionals have a lot of extra support built into the language. Some of the examples below might look very simple if you've been writing Swift for a while, but it's important to make sure you know all of these concepts well, as we'll be using them again and again throughout the book.

if let

Optional binding with if let is just a short step away from the switch statement above:

```
var array = ["one", "two", "three", "four"]
if let idx = array.index(of: "four") {
    array.remove(at: idx)
}
```

An optional binding with if let can have boolean clauses as well. So suppose you didn't want to remove the element if it happened to be the first one in the array:

```
if let idx = array.index(of: "four"), idx != array.startIndex {
    array.remove(at: idx)
}
```

You can also bind multiple values in the same if statement. What's more is that later entries can rely on the earlier ones being successfully unwrapped. This is very useful when you want to make multiple calls to functions that return optionals themselves. For example, these URL and UIImage initializers are all "failable" — that is, they can return nil — if your URL is malformed, or if the data isn't an image. The Data initializer can throw an error, and by using try?, we can convert it into an optional as well. All three can be chained together, like this:

```
let urlString = "https://www.objc.io/logo.png"
if let url = URL(string: urlString),
    let data = try? Data(contentsOf: url),
    let image = UIImage(data: data)
{
    let view = UIImageView(image: image)
    PlaygroundPage.current.liveView = view
}
```

Any part of a multi-variable let can have a boolean clause as well:

```
if let url = URL(string: urlString), url.pathExtension == "png",
    let data = try? Data(contentsOf: url),
    let image = UIImage(data: data)
{
    let view = UIImageView(image: image)
}
```

If you need to perform a check *before* performing various if let bindings, you can supply a leading boolean condition. Suppose you're transitioning to a new scene in an iOS app and want to check the segue identifier before casting to a specific kind of view controller:

```
if segue.identifier == "showUserDetailsSegue",
    let userDetailVC = segue.destination
        as? UserDetailViewController
{
    userDetailVC.screenName = "Hello"
}
```

You can also mix and match optional bindings, boolean clauses, and case let bindings within the same if statement.

This kind of if let binding is a good match for the Scanner type in Foundation, which returns a boolean value to indicate whether or not it successfully scanned something, after which you can unwrap the result:

```
let scanner = Scanner(string: "lisa123")
var username: NSString?
let alphas = CharacterSet.alphanumerics
if scanner.scanCharacters(from: alphas, into: &username),
    let name = username {
    print(name)
}
```

while let

Very similar to the if let statement is while let — a loop that only terminates when its condition returns nil.

The standard library's readLine function returns an optional string from the standard input. Once the end of input is reached, it returns nil. So to implement a very basic equivalent of the Unix cat command, you use while let:

```
while let line = readLine() {
    print(line)
}
```

Similar to if let, you can always add a boolean clause to your optional binding. So if you want to terminate this loop on either EOF or a blank line, add a clause to detect an empty string. Note that once the condition is false, the loop is terminated (you might mistakenly think the boolean condition functions like a filter):

```
while let line = readLine(), !line.isEmpty {
    print(line)
}
```

As we saw in the chapter on collection protocols, the for x in seq loop requires seq to conform to Sequence. This protocol provides a makeIterator method that returns an iterator, which in turn has a next method. next returns values until the sequence is exhausted, and then it returns nil. while let is ideal for this:

```
let array = [1, 2, 3]
var iterator = array.makeIterator()
while let i = iterator.next() {
    print(i, terminator: " ")
} // 1 2 3
```

So given that for loops are really just while loops, it's not surprising that they also support boolean clauses, albeit with a where keyword:

```
for i in 0..<10 where i % 2 == 0 {
    print(i, terminator: " ")
} // 0 2 4 6 8
```

Note that the where clause above doesn't work like the boolean clause in a while loop. In a while loop, iteration stops once the value is false, whereas in a for loop, it functions like filter. If we rewrite the above for loop using while, it looks like this:

```
var iterator2 = (0..<10).makeIterator()
while let i = iterator2.next() {
    if i % 2 == 0 {
        print(i)
    }
}
```

This feature of for loops avoids a particularly strange bug with variable capture that can occur in other languages. Consider the following code, written in Ruby:

```
functions = []
for i in 1..3
    functions.push(lambda { i })
end
for f in functions
    print "#{f.call()}"
end
```

Ruby lambdas are like Swift's closure expressions, and as with Swift, they capture local variables. So the above code loops from 1 to 3 — adding a closure to the array that captures i — and will print out the value of i when called. Then it loops over that array, calling each of the closures. What do you think will be printed out? If you're on a Mac, you can try it out by pasting the above into a file and running ruby on it from the command line.

If you run it, you'll see it prints out three 3s in a row. Even though i held a different value when each closure was created, they all captured the *same* i variable. And when you call them, i now has the value 3 — its value at the end of the loop.

Now for a similar Swift snippet:

```
var functions: [() -> Int] = []
for i in 1...3 {
    functions.append { i }
}
for f in functions {
    print("\(f())", terminator: " ")
} // 1 2 3
```

The output: 1, 2, and 3. This makes sense when you realize for...in is really while let. To make the correspondence even clearer, imagine there *wasn't* a while let, and that you had to use an iterator without it:

```
var functions1: [() -> Int] = []
var iterator1 = (1...3).makeIterator()
var current1: Int? = iterator1.next()
while current1 != nil {
    let i = current1!
    functions1.append { i }
    current1 = iterator1.next()
}
```

This makes it easy to see that i is a fresh local variable in every iteration, so the closure captures the correct value even when a *new* local i is declared on subsequent iterations.

By contrast, the Ruby code is more along the lines of the following:

```
var functions2: [() -> Int] = []

do {
    var iterator2 = (1...3).makeIterator()
    var i: Int
    var current: Int? = iterator2.next()
    while current != nil {
        i = current!
        functions2.append { i }
        current = iterator2.next()
    }
}
```

Here, i is declared *outside* the loop — and reused — so every closure captures the same i. If you run each of them, they'll all return 3. The do is there because, despite i being declared outside the loop, it's still scoped in such a way that it isn't *accessible* outside that loop — it's sandwiched in a narrow outer shell.

> *Most Swift programmers know do blocks as part of the do { try ... } catch { ... } construct used for error handling. We'll discuss this usage in the chapter on error handling. But do blocks can also be used on their own to introduce a new scope. Swift landed on this syntax because C's keywordless { ... } syntax is already used for closure expressions in Swift.*

Python behaves like Ruby. C# had the same behavior too — until C# 5, when it was decided that this behavior was dangerous enough to justify a breaking change in order to make it work like Swift.

Doubly Nested Optionals

This is a good time to point out that the type an optional wraps can itself be optional, which leads to optionals nested inside optionals. To see why this isn't just a strange edge case or something the compiler should automatically coalesce, suppose you have an array of strings of numbers, which you want to convert into integers. You might run them through a map to convert them:

```
let stringNumbers = ["1", "2", "three"]
let maybeInts = stringNumbers.map { Int($0) } // [Optional(1), Optional(2), nil]
```

You now have an array of Optional<Int> — i.e. Int? — because Int.init(String) is failable, since the string might not contain a valid integer. Here, the last entry will be a nil, since "three" isn't an integer.

When looping over the array with for, you'd rightly expect that each element would be an optional integer, because that's what maybeInts contains:

```
for maybeInt in maybeInts {
    // maybeInt is an Int?
    // Two numbers and a `nil`
}
```

Now consider that the implementation of for...in is shorthand for the while loop technique above. What's returned from iterator.next() would be an Optional<Optional<Int>> — or Int?? — because next wraps each element in the sequence inside an optional. The while let unwraps it to check it isn't nil, and while it's non-nil, binds the unwrapped value and runs the body:

```
var iterator = maybeInts.makeIterator()
while let maybeInt = iterator.next() {
    print(maybeInt, terminator: " ")
}
// Optional(1) Optional(2) nil
```

When the loop gets to the final element — the nil from "three" — what's returned from next is a non-nil value: .some(nil). It unwraps this and binds what's inside (a nil) to maybeInt. Without nested optionals, this wouldn't be possible.

By the way, if you ever want to loop over only the non-nil values with for, you can use case pattern matching:

```
for case let i? in maybeInts {
    // i will be an Int, not an Int?
    print(i, terminator: " ")
}
// 1 2

// Or only the nil values:
```

```
for case nil in maybeInts {
    // Will run once for each nil
    print("No value")
}
// No value
```

This uses a “pattern” of x?, which only matches non-nil values. This is shorthand for .some(x), so the loop could be written like this:

```
for case let .some(i) in maybeInts {
    print(i)
}
```

This case-based pattern matching is a way to apply the same rules that work in switch statements to if, for, and while. It’s most useful with optionals, but it also has other applications — for example:

```
let j = 5
if case 0..<10 = j {
    print("\(j) within range")
} // 5 within range
```

Since case matching is extensible via overloading the ~= operator, this means you can extend if case and for case in various interesting ways:

```
struct Pattern {
    let s: String
    init(_ s: String) { self.s = s }
}

func ~=(pattern: Pattern, value: String) -> Bool {
    return value.range(of: pattern.s) != nil
}

let s = "Taylor Swift"
if case Pattern("Swift") = s {
    print("\(String(reflecting: s)) contains \"Swift\"")
}
// "Taylor Swift" contains "Swift"
```

This has incredible potential, but you need to be careful, as it's very easy to accidentally write ~= operators that match a little too much. On that note, inserting the following into a common bit of library code would probably be a good April Fools' joke:

```
func ~=<T, U>(_: T, _: U) -> Bool { return true }
```

This code will make every case match (unless a more specific version of ~= is already defined).

if var and while var

Instead of let, you can use var with if, while, and for. This allows you to mutate the variable inside the statement body:

```
let number = "1"
if var i = Int(number) {
    i += 1
    print(i)
} // 2
```

But note that i will be a local copy; any changes to i won't affect the value inside the original optional. Optionals are value types, and unwrapping them copies the value inside.

Scoping of Unwrapped Optionals

Sometimes it feels limiting to only have access to an unwrapped variable within the if block it has defined. For example, take the first property on arrays — a property that returns an optional of the first element, or nil when the array is empty. This is convenient shorthand for the following common bit of code:

```
let array = [1,2,3]
```

```
if !array.isEmpty {
    print(array[0])
}
// Outside the block, the compiler can't guarantee that array[0] is valid
```

Using the first property, you *have* to unwrap the optional in order to use it — you can't accidentally forget:

```
if let firstElement = array.first {
    print(firstElement)
}
// Outside the block, you can't use firstElement
```

The unwrapped value is only available inside the if let block. This is impractical if the purpose of the if statement is to exit early from a function when some condition isn't met. This early exit can help avoid annoying nesting or repeated checks later on in the function. You might write the following:

```
func doStuff(withArray a: [Int]) {
    if a.isEmpty {
        return
    }
    // Now use a[0] or a.first! safely
}
```

Here, if let binding wouldn't work because the bound variable wouldn't be in scope after the if block. But you can nevertheless be sure the array will contain at least one element, so force-unwrapping the first element is safe, even if the syntax is still unappealing.

One option for using an unwrapped optional outside the scope it was bound in is to rely on Swift's deferred initialization capabilities. Consider the following example, which reimplements part of the pathExtension property from URL and NSString:

```
extension String {
    var fileExtension: String? {
        let period: String.Index
        if let idx = index(of: ".") {
            period = idx
        } else {
            return nil
        }
        let extensionStart = index(after: period)
        return String(self[extensionStart...])
    }
}

"hello.txt".fileExtension // Optional("txt")
```

The compiler checks your code to confirm there are only two possible paths: one in which the function returns early, and another where period is properly initialized. There's no way period could be nil (it isn't optional) or uninitialized (Swift won't let you use a variable that hasn't been initialized). So after the if statement, the code can be written without you having to worry about optionals at all.

However, the two previous examples are pretty ugly. Really, what's needed is some kind of if not let — which is exactly what guard let does:

```
func doStuff(withArray a: [Int]) {
    guard let firstElement = a.first else {
        return
    }
    // firstElement is unwrapped here
}
```

And the second example becomes much clearer:

```
extension String {
    var fileExtension: String? {
        guard let period = index(of: ".") else {
            return nil
        }
        let extensionStart = index(after: period)
        return String(self[extensionStart...])
    }
}
```

Anything can go in the else clause here, including multiple statements just like an if ... else. The only requirement is that the else block must leave the current scope. That might mean return, or it might mean calling fatalError (or any other function that returns Never). If the guard were in a loop, break or continue would also be allowed.

A function that has the return type Never *signals to the compiler that it'll never return. There are two common types of functions that do this: those that abort the program, such as* fatalError*; and those that run for the entire lifetime of the program, like* dispatchMain*. The compiler uses this information for its control flow diagnostics. For example, the* else *branch of a guard statement must either exit the current scope or call one of these never-returning functions.*

Never *is what's called an uninhabited type. It's a type that has no valid values and thus can't be constructed. Its only purpose is its signaling role for the compiler. A function declared to return an uninhabited type can never return normally. In Swift, an uninhabited type is implemented as an* enum *that has no cases:*

```
public enum Never { }
```

You won't usually need to define your own never-returning functions unless you write a wrapper for fatalError *or* preconditionFailure*. One interesting use case is while you're writing new code: say you're working on a complex* switch *statement, gradually filling in all the cases, and the compiler is bombarding you with error messages for empty case labels or missing return values, while all you'd like to do is concentrate on the one case you're working on. In this situation, a few carefully placed calls to* fatalError() *can do wonders to silence* *the compiler. Consider writing a function called unimplemented() in order to better communicate the temporary nature of these calls:*

```
func unimplemented() -> Never {
    fatalError("This code path is not implemented yet.")
}
```

Swift meticulously distinguishes between different kinds of "nothingness." In addition to nil and Never, there's also Void, which is just another way of writing an empty tuple:

```
public typealias Void = ()
```

The most common use of Void or () is in the types of functions that don't return anything, but it has other applications too. For example, consider a reactive programming framework that models event streams with an Observable<T> type, where T describes the payload type of the emitted event. A text field object might provide an Observable<String> that fires an event every time the user edits the text. Similarly, a button object sends an event when the user taps the button, but it has no additional payload to send — its event stream should have the type Observable<()>.

As David Smith put it, Swift makes a careful distinction between the "absence of a thing" (nil), the "presence of nothing" (Void), and a "thing which cannot be" (Never).

Of course, guard isn't limited to binding. guard can take any condition you might find in a regular if statement, so the empty array example could be rewritten with it:

```
func doStuff2(withArray a: [Int]) {
    guard !a.isEmpty else { return }
    // Now use a[0] or a.first! safely
}
```

Unlike the optional binding case, this guard isn't a big win — in fact, it's slightly more verbose than the original return. But it's still worth considering doing this with any early exit situation. For one, sometimes (though not in this case) the inversion of the boolean condition can make things clearer. Additionally, guard is a clear signal when reading the code; it says: "We only continue if the following condition holds." Finally, the Swift compiler will check that you're definitely exiting the current scope and raise a compilation error if you don't. For this reason, we'd suggest using guard even when an if would do.

Optional Chaining

In Objective-C, sending a message to nil is a no-op. In Swift, the same effect can be achieved via "optional chaining":

```
delegate?.callback()
```

Unlike with Objective-C, though, the Swift compiler will force you to acknowledge that the receiver could be nil. The question mark is a clear signal to the reader that the method might not be called.

When the method you call via optional chaining returns a result, that result will also be optional. Consider the following code to see why this must be the case:

```
let str: String? = "Never say never"
// We want upper to be the uppercase string
let upper: String
if str != nil {
    upper = str!.uppercased()
} else {
    // No reasonable action to take at this point
    fatalError("No idea what to do now...")
}
```

If str is non-nil, upper will have the desired value. But if str is nil, then upper can't be set to a value. So in the optional chaining case, result *must* be optional, in order to account for the possibility that str could've been nil:

```
let result = str?.uppercased() // Optional("NEVER SAY NEVER")
```

As the name implies, you can chain calls on optional values:

```
let lower = str?.uppercased().lowercased() // Optional("never say never")
```

This might look a bit surprising. Didn't we just say that the result of optional chaining is an optional? So why don't you need a ?. after uppercased()? This is because optional chaining is a "flattening" operation. If str?.uppercased() returned an optional and you called ?.lowercased() on it, then logically you'd get an optional optional. But you just want a regular optional, so instead we write the second chained call without the question mark to represent the fact that the optionality is already captured.

On the other hand, if the uppercased method itself returned an optional, then you *would* need to add a second ? to express that you were chaining *that* optional. For example, let's extend the Int type with a computed property named half. This property returns the result of dividing the integer by two, but only if the number is big enough to be divided. When the number is smaller than two, it returns nil:

```
extension Int {
    var half: Int? {
        guard self < -1 || self > 1 else { return nil }
        return self / 2
    }
}
```

Because calling half returns an optional result, we need to keep putting in ? when calling it repeatedly. After all, at every step, the function might return nil:

```
20.half?.half?.half // Optional(2)
```

Notice that the compiler is still smart enough to flatten the result type for us. The type of the expression above is Int? and not Int???, as you might expect. The latter would give you more information — namely, which part of the chain failed — but it would also make it a lot more cumbersome to deal with the result, destroying the convenience optional chaining adds in the first place.

Optional chaining also applies to subscripts — for example:

```
let dictOfArrays = ["nine": [0, 1, 2, 3]]
dictOfArrays["nine"]?[3] // Optional(3)
```

Additionally, you can use optional chaining to call optional functions:

```
let dictOfFunctions: [String: (Int, Int) -> Int] = [
    "add": (+),
    "subtract": (-)
]
dictOfFunctions["add"]?(1, 1) // Optional(2)
```

This is handy in typical callback situations where a class stores a callback function in order to inform its owner when an event occurs. Consider a TextField class:

```
class TextField {
    private(set) var text = ""
    var didChange: ((String) -> ())?

    private func textDidChange(newText: String) {
        text = newText
        // Trigger callback if non-nil
        didChange?(text)
    }
}
```

The didChange property stores a callback function, which the text field calls every time the user edits the text. Because the text field's owner doesn't have to register a callback, the property is optional; its initial value is nil. When the time comes to invoke the callback (in the textDidChange method above), optional chaining lets us do so in a very compact way.

You can even assign *through* an optional chain. Suppose you have an optional variable, and if it's non-nil, you wish to update one of its properties:

```
struct Person {
    var name: String
    var age: Int
}

var optionalLisa: Person? = Person(name: "Lisa Simpson", age: 8)
// Increment age if non-nil
if optionalLisa != nil {
    optionalLisa!.age += 1
}
```

This is rather verbose and ugly. Note that you can't use optional binding in this case. Since Person is a struct and thus a value type, the bound variable is a local copy of the original value; mutating the former won't change the latter:

```
if var lisa = optionalLisa {
    // Mutating lisa doesn't change optionalLisa
    lisa.age += 1
}
```

This *would* work if Person were a class. We'll talk more about the differences between value and reference types in the structs and classes chapter. Regardless, it's still too verbose. Instead, you can assign to the chained optional value, and if it isn't nil, the assignment will go through:

```
optionalLisa?.age += 1
```

A weird (but logical) edge case of this feature is that it works for direct assignment to optional values. This is perfectly valid:

```
var a: Int? = 5
a? = 10
a // Optional(10)

var b: Int? = nil
b? = 10
b // nil
```

Notice the subtle difference between a = 10 and a? = 10. The former assigns a new value unconditionally, whereas the latter only performs the assignment if the value of a is non-nil *before* the assignment.

The nil-Coalescing Operator

Often you want to unwrap an optional, replacing nil with some default value. This is a job for the nil-coalescing operator:

```
let stringteger = "1"
let number = Int(stringteger) ?? 0
```

So if the string can be converted to an integer, number will be that integer, unwrapped. If it isn't, and Int.init returns nil, the default value of 0 will be substituted. So lhs ?? rhs is analogous to the code lhs != nil ? lhs! : rhs.

"Big deal!" Objective-C developers might say. "We've had the ?: for ages." And ?? is very similar to Objective-C's ?:. But there are some differences, so it's worth stressing an important point when thinking about optionals in Swift: optionals are *not* pointers.

Yes, you'll often encounter optionals combined with references when dealing with Objective-C libraries. But optionals, as we've seen, can also wrap value types. So number in the above example is just an Int, not an NSNumber.

Through the use of optionals, you can guard against much more than just null pointers. Consider the case where you want to access the first value of an array — but in case the array is empty, you want to provide a default:

```
let array = [1,2,3]
!array.isEmpty ? array[0] : 0
```

Because Swift arrays provide a `first` property that's `nil` if the array is empty, you can use the nil-coalescing operator instead:

```
array.first ?? 0 // 1
```

This is cleaner and clearer — the intent (grab the first element in the array) is up front, with the default tacked on the end, joined with a ?? that signals "this is a default value." Compare this with the ternary version, which starts first with the check, then the value, then the default. And the check is awkwardly negated (the alternative being to put the default in the middle and the actual value on the end). And, as is the case with optionals, it's impossible to forget that first is optional and accidentally use it without the check, because the compiler will stop you if you try.

Whenever you find yourself guarding a statement with a check to make sure the statement is valid, it's a good sign optionals would be a better solution. Suppose that instead of an empty array, you're checking a value that's within the array bounds:

```
array.count > 5 ? array[5] : 0 // 0
```

Unlike `first` and `last`, getting an element out of an array by its index doesn't return an `Optional`. But it's easy to extend `Array` to add this functionality:

```
extension Array {
    subscript(guarded idx: Int) -> Element? {
        guard (startIndex..<endIndex).contains(idx) else {
            return nil
        }
        return self[idx]
    }
}
```

This now allows you to write the following:

```
array[guarded: 5] ?? 0 // 0
```

Coalescing can also be chained — so if you have multiple possible optionals and you want to choose the first non-nil value, you can write them in sequence:

```
let i: Int? = nil
let j: Int? = nil
let k: Int? = 42
i ?? j ?? k ?? 0 // 42
```

Sometimes, you might have multiple optional values and you want to choose between them in an order, but you don't have a reasonable default if they're all nil. You can still use ?? for this, but if the final value is also optional, the full result will be optional:

```
let m = i ?? j ?? k
type(of: m) // Optional<Int>
```

This is often useful in conjunction with if let. You can think of this like an "or" equivalent of if let:

```
if let n = i ?? j { // similar to if i != nil || j != nil
    print(n)
}
```

If you think of the ?? operator as similar to an "or" statement, you can think of an if let with multiple clauses as an "and" statement:

```
if let n = i, let m = j { }
// similar to if i != nil && j != nil
```

Because of this chaining, if you're ever presented with a doubly nested optional and want to use the ?? operator, you must take care to distinguish between a ?? b ?? c (chaining) and (a ?? b) ?? c (unwrapping the inner and then outer layers):

```
let s1: String?? = nil // nil
(s1 ?? "inner") ?? "outer" // inner
let s2: String?? = .some(nil) // Optional(nil)
(s2 ?? "inner") ?? "outer" // outer
```

Using Optionals with String Interpolation

You may have noticed that the compiler emits a warning when you print an optional value or use one in a string interpolation expression:

```
let bodyTemperature: Double? = 37.0
let bloodGlucose: Double? = nil
print(bodyTemperature) // Optional(37.0)
// Warning: Expression implicitly coerced from 'Double?' to Any
print("Blood glucose level: \(bloodGlucose)") // Blood glucose level: nil
// Warning: String interpolation produces a debug description
// for an optional value; did you mean to make this explicit?
```

This is generally helpful because it's all too easy for a stray "Optional(...)" or "nil" to accidentally creep into a text displayed to the user. You should never use optionals directly in user-facing strings and always unwrap them first. Since string interpolation is defined for all types (including Optional), the compiler can't make this a hard error, though — a warning is really the best it can do.

Sometimes you may *want* to use an optional in a string interpolation — to log its value for debugging, for example — and in that case, the warnings can become annoying. The compiler offers several fix-its to silence the warning: add an explicit cast with as Any, force-unwrap the value with ! (if you're certain it can't be nil), wrap it in String(describing: ...), or provide a default value with the nil-coalescing operator.

The latter is often a quick and pretty solution, but it has one drawback: the types on both sides of the ?? expression must match, so the default value you provide for a Double? must be of type Double. Since the ultimate goal is to turn the expression into a string, it'd be convenient if you could provide a string as a default value in the first place.

Swift's ?? operator doesn't support this kind of type mismatch — after all, what would the type of the expression be if the two sides didn't have a common base type? But it's easy to add your own operator, especially for the purpose of working with optionals in string interpolation. Let's name it ???:

```
infix operator ???: NilCoalescingPrecedence

public func ???<T>(optional: T?, defaultValue: @autoclosure () -> String)
    -> String
{
    switch optional {
    case let value?: return String(describing: value)
    case nil: return defaultValue()
    }
}
```

This takes any optional T? on the left side and a string on the right. If the optional is non-nil, we unwrap it and return its string description. Otherwise, we return the default string that was passed in. The @autoclosure annotation makes sure that we only evaluate the second operand when needed. In the chapter on functions, we'll go into this in more detail.

Now we can write the following and we won't get any compiler warnings:

```
print("Body temperature:\(bodyTemperature ??? "n/a")")
// Body temperature: 37.0
```

```
print("Blood glucose level:\(bloodGlucose ??? "n/a")")
// Blood glucose level: n/a
```

Optional map

Let's say we have an array of characters, and we want to turn the first element into a string:

```
let characters: [Character] = ["a", "b", "c"]
String(characters[0]) // a
```

If it's possible for characters to be empty, we can use an if let to create the string only if the array is non-empty:

```
var firstCharAsString: String? = nil
if let char = characters.first {
    firstCharAsString = String(char)
}
```

So now, if the array contains at least one element, firstCharAsString will contain that element as a String. But if it doesn't, firstCharAsString will be nil.

This pattern — take an optional, and transform it if it isn't nil — is common enough that there's a method on optionals to do this. It's called map, and it takes a closure that represents how to transform the contents of the optional. Here's the above function, rewritten using map:

```
let firstChar = characters.first.map { String($0) } // Optional("a")
```

This map is, of course, very similar to the map on arrays or other sequences. But instead of operating on a sequence of values, it operates on just one: the possible one inside the optional. You can think of optionals as being a collection of either zero or one values, with map either doing nothing to zero values or transforming one.

Given the similarities, the standard library's implementation of optional map looks a lot like collection map:

```
extension Optional {
    func map<U>(transform: (Wrapped) -> U) -> U? {
        if let value = self {
            return transform(value)
        }
        return nil
    }
}
```

An optional map is especially nice when you already want an optional result. Suppose you wanted to write another variant of reduce for arrays. Instead of taking an initial value, it uses the first element in the array (in some languages, this might be called reduce1, but we'll call it reduce and rely on overloading):

Because of the possibility that the array might be empty, the result needs to be optional — without an initial value, what else could it be? You might write it like this:

```
extension Array {
    func reduce(_ nextPartialResult: (Element, Element) -> Element) -> Element? {
        // first will be nil if the array is empty
        guard let fst = first else { return nil }
        return dropFirst().reduce(fst, nextPartialResult)
    }
}
```

You can use it like this:

```
[1, 2, 3, 4].reduce(+) // Optional(10)
```

Since optional map returns nil if the optional is nil, reduce could be rewritten using a single return statement (and no guard):

```
extension Array {
    func reduce_alt(_ nextPartialResult: (Element, Element) -> Element)
        -> Element?
    {
        return first.map {
            dropFirst().reduce($0, nextPartialResult)
        }
    }
}
```

Optional flatMap

As we saw in the built-in collections chapter, it's common to want to map over a collection with a function that returns a collection, but collect the results as a single array rather than an array of arrays.

Similarly, if you want to perform a map on an optional value, but your transformation function also has an optional result, you'll end up with a doubly nested optional. An example of this is when you want to fetch the first element of an array of strings as a number, using first on the array and then map to convert it to a number:

```
let stringNumbers = ["1", "2", "3", "foo"]
let x = stringNumbers.first.map { Int($0) } // Optional(Optional(1))
```

The problem is that since map returns an optional (first might have been nil) and Int(someString) returns an optional (the string might not be an integer), the type of x will be Int??.

flatMap will instead flatten the result into a single optional. As a result, y will be of type Int?:

```
let y = stringNumbers.first.flatMap { Int($0) } // Optional(1)
```

Instead, you could've written this with if let, because values that are bound later can be computed from earlier ones:

```
if let a = stringNumbers.first, let b = Int(a) {
    print(b)
} // 1
```

This shows that flatMap and if let are very similar. Earlier in this chapter, we saw an example that uses a multiple-if-let statement. We can rewrite it using map and flatMap instead:

```
let urlString = "https://www.objc.io/logo.png"
let view = URL(string: urlString)
    .flatMap { try? Data(contentsOf: $0) }
    .flatMap { UIImage(data: $0) }
    .map { UIImageView(image: $0) }

if let view = view {
    PlaygroundPage.current.liveView = view
}
```

Optional chaining is also very similar to flatMap: i?.advance(by: 1) is essentially equivalent to i.flatMap { $0.advance(by: 1) }.

Since we've shown that a multiple-if-let statement is equivalent to flatMap, we could implement one in terms of the other:

```
extension Optional {
    func flatMap<U>(transform: (Wrapped) -> U?) -> U? {
        if let value = self, let transformed = transform(value) {
            return transformed
        }
        return nil
    }
}
```

Filtering Out nils with flatMap

If you have a sequence and it contains optionals, you might not care about the nil values. In fact, you might just want to ignore them.

Suppose you wanted to process only the numbers in an array of strings. This is easily done in a for loop using optional pattern matching:

```
let numbers = ["1", "2", "3", "foo"]
var sum = 0
for case let i? in numbers.map({ Int($0) }) {
    sum += i
}
sum // 6
```

You might also want to use ?? to replace the nils with zeros:

```
numbers.map { Int($0) }.reduce(0) { $0 + ($1 ?? 0) } // 6
```

But really, you just want a version of map that filters out nil and unwraps the non-nil values. Enter the standard library's overload of flatMap on sequences, which does exactly that:

```
numbers.flatMap { Int($0) }.reduce(0, +) // 6
```

We've already seen two flattening maps: flattening a sequence mapped to arrays, and flattening an optional mapped to an optional. This is a hybrid of the two: flattening a sequence mapped to an optional.

This makes sense if we return to our analogy of an optional being a collection of zero or one thing(s). If that collection were an array, flatMap would be exactly what we want.

To implement our own version of this operation, let's first define a flatten that filters out nil values and returns an array of non-optionals:

```
func flatten<S: Sequence, T>
    (source: S) -> [T] where S.Element == T? {
    let filtered = source.lazy.filter { $0 != nil }
    return filtered.map { $0! }
}
```

Ewww, a free function? Why no protocol extension? Unfortunately, there's no way to constrain an extension on Sequence to only apply to sequences of optionals. You'd need a two-placeholder clause (one for S, and one for T, as given here), and protocol extensions currently don't support this.

Nonetheless, it does make flatMap simple to write:

```
extension Sequence {
    func flatMap<U>(transform: (Element) -> U?) -> [U] {
        return flatten(source: self.lazy.map(transform))
    }
}
```

In both these functions, we used lazy to defer actual creation of the array until the last moment. This is possibly a micro-optimization, but it might be worthwhile for larger arrays. Using lazy saves the allocation of multiple buffers that would otherwise be needed to write the intermediary results into.

Equating Optionals

Often, you don't care whether a value is nil or not — just whether it contains (if non-nil) a certain value:

```
let regex = "^Hello$"
// ...
if regex.first == "^" {
    // match only start of string
}
```

In this case, it doesn't matter if the value is nil or not — if the string is empty, the first character can't be a caret, so you don't want to run the block. But you still want the protection and simplicity of first. The alternative, if !regex.isEmpty && regex[regex.startIndex] == "^", is horrible.

The code above relies on two things to work. First, there's a version of == that takes two optionals, with an implementation something like this:

```
func ==<T: Equatable>(lhs: T?, rhs: T?) -> Bool {
    switch (lhs, rhs) {
    case (nil, nil): return true
    case let (x?, y?): return x == y
    case (_?, nil), (nil, _?): return false
    }
}
```

This overload *only* works on optionals of equatable types. Given this, there are four possibilities: they're both nil, or they both have a value, or either one or the other is nil. The switch exhaustively tests all four possibilities (hence no need for a default clause). It defines two nils to be equal to each other, nil to never be equal to non-nil, and two non-nil values to be equal if their unwrapped values are equal.

But this is only half the story. Notice that we did *not* have to write the following:

```
if regex.first == Optional("^") { // or: == .some("^")
    // Match only start of string
}
```

This is because whenever you have a non-optional value, Swift will always be willing to upgrade it to an optional value in order to make the types match.

This implicit conversion is incredibly useful for writing clear, compact code. Suppose there was no such conversion, but to make things nice for the caller, you wanted a version of == that worked between both optional and non-optional types. You'd have to write three separate versions:

```
// Both optional
func == <T: Equatable>(lhs: T?, rhs: T?) -> Bool
```

```
// lhs non-optional
func == <T: Equatable>(lhs: T, rhs: T?) -> Bool
// rhs non-optional
func == <T: Equatable>(lhs: T?, rhs: T) -> Bool
```

But instead, only the first version is necessary, and the compiler will convert to optionals where necessary.

In fact, we've been relying on this throughout the book. For example, when we implemented optional map, we transformed the inner value and returned it. But the return value of map is optional. The compiler automatically converted the value for us — we didn't have to write `return Optional(transform(value))`.

Swift code constantly relies on this implicit conversion. For example, dictionary subscript lookup by key returns an optional (the key might not be present). But it also takes an optional on assignment — subscripts have to both take and receive the same type. Without implicit conversion, you'd have to write `myDict["someKey"] = Optional(someValue)`.

Incidentally, if you're wondering what happens to dictionaries with key-based subscript assignment when you assign a `nil` value, the answer is that the key is removed. This can be useful, but it also means you need to be a little careful when dealing with a dictionary with an optional value type. Consider this dictionary:

```
var dictWithNils: [String: Int?] = [
    "one": 1,
    "two": 2,
    "none": nil
]
```

The dictionary has three keys, and one of them has a value of `nil`. Suppose we wanted to set the value of the `"two"` key to `nil` as well. This will *not* do that:

```
dictWithNils["two"] = nil
dictWithNils // ["none": nil, "one": Optional(1)]
```

Instead, it'll *remove* the "two" key.

To change the value for the key, you'd have to write one of the following (they all work, so choose whichever you feel is clearer):

```
dictWithNils["two"] = Optional(nil)
dictWithNils["two"] = .some(nil)
dictWithNils["two"]? = nil
dictWithNils // ["none": nil, "one": Optional(1), "two": nil]
```

Note that the third version above is slightly different than the other two. It works because the "two" key is already in the dictionary, so it uses optional chaining to set its value if successfully fetched. Now try this with a key that isn't present:

```
dictWithNils["three"]? = nil
dictWithNils.index(forKey: "three") // nil
```

You can see that nothing would be updated/inserted.

Equatable and ==

Even though optionals have an == operator, this doesn't mean they can conform to the Equatable protocol. This subtle but important distinction will hit you in the face if you try and do the following:

```
// Two arrays of optional integers
let a: [Int?] = [1, 2, nil]
let b: [Int?] = [1, 2, nil]

// Error: binary operator '==' cannot be applied to two [Int?] operands
a == b
```

The problem is that the == operator for arrays requires the elements of the array to be equatable:

```
func ==<Element : Equatable>(lhs: [Element], rhs: [Element]) -> Bool
```

Optionals don't conform to Equatable — that would require they implement == for any kind of type they contain, and they only can if that type is itself equatable. In the future, Swift will support conditional protocol conformance, and only then will we be able to write something like this:

```
extension Optional: Equatable where Wrapped: Equatable {
    // No need to implement anything; == is already implemented so long
    // as this condition is met.
}
```

In the meantime, you could implement a version of == for arrays of optionals, like so:

```
func ==<T: Equatable>(lhs: [T?], rhs: [T?]) -> Bool {
    return lhs.elementsEqual(rhs) { $0 == $1 }
}
```

Comparing Optionals

Similar to ==, there used to be implementations of <, >, <=, and >= for optionals. For Swift 3.0, these comparison operators were removed for optionals because they can easily yield unexpected results.

For example, nil < .some(_) would return true. In combination with higher-order functions or optional chaining, this can be very surprising. Consider the following example:

```
let temps = ["-459.67", "98.6", "0", "warm"]
let belowFreezing = temps.filter { Double($0) < 0 }
```

Because Double("warm") will return nil and nil is less than 0, it'll be included in the belowFreezing temperatures. This is unexpected indeed.

If you need inequality relations between optionals, you now have to unwrap the values first and thereby explicitly state how nil values should be handled. We'll show an example of this in the chapter on functions, where we define a generic function that "lifts" a regular comparison function into the domain of optionals.

When to Force-Unwrap

Given all these techniques for cleanly unwrapping optionals, when should you use !, the force-unwrap operator? There are many opinions on this scattered throughout the Internet, such as "never," "whenever it makes the code clearer," and "when you can't avoid it." We propose the following rule, which encompasses most of them:

> *Use ! when you're so certain that a value won't be nil that you want your program to crash if it ever is.*

As an example, take the implementation of flatten:

```
func flatten<S: Sequence, T>
    (source: S) -> [T] where S.Element == T? {
    let filtered = source.lazy.filter { $0 != nil }
    return filtered.map { $0! }
}
```

Here, there's no possible way that $0! inside map will ever hit a nil, since the nil elements were all filtered out in the preceding step. This function could certainly be written to eliminate the force-unwrap operator by looping over the array and adding non-nil values into an array. But the filter/map version is cleaner and probably clearer, so the ! is justified.

However, these cases are pretty rare. If you have full mastery of all the unwrapping techniques described in this chapter, chances are there's a better way. Whenever you do find yourself reaching for !, it's worth taking a step back and wondering if there really is no other option.

As another example, consider the following code that fetches all the keys in a dictionary with values matching a certain condition:

```
let ages = [
    "Tim": 53, "Angela": 54, "Craig": 44,
    "Jony": 47, "Chris": 37, "Michael": 34,
]
ages.keys
    .filter { name in ages[name]! < 50 }
    .sorted()
// ["Chris", "Craig", "Jony", "Michael"]
```

Again, the ! is perfectly safe here — since all the keys came from the dictionary, there's no possible way in which a key could be missing from the dictionary.

But you could also rewrite the statement to eliminate the need for a force-unwrap altogether. Using the fact that dictionaries present themselves as sequences of key-value pairs, you could just filter this sequence and then run it through a map to remove the value:

```
ages.filter { (_, age) in age < 50 }
    .map { (name, _) in name }
    .sorted()
// ["Chris", "Craig", "Jony", "Michael"]
```

This version even has a possible performance benefit because it avoids unnecessary key lookups.

Nonetheless, sometimes life hands you an optional, and you know *for certain* that it isn't nil. So certain are you of this that you'd *rather* your program crash than continue, because it'd mean a very nasty bug in your logic. Better to trap than to continue under those circumstances, so ! acts as a combined unwrap-or-error operator in one handy character. This approach is often a better move than just using the nil chaining or coalescing operators to sweep theoretically impossible situations under the carpet.

The Swift compiler is a little too trigger-happy with its fix-it suggestions in this area for our taste. It often suggests adding a force-unwrap when it encounters an optional in a place where it expects a non-optional value, presumably because it's the only "fix" it can insert without requesting further input from you (like a default value). Things will almost certainly go bad if you apply these fix-its blindly.

Improving Force-Unwrap Error Messages

That said, even when you're force-unwrapping an optional value, you have options other than using the ! operator. When your program does error, you don't get much by way of description as to why in the output log.

Chances are, you'll leave a comment as to why you're justified in force-unwrapping. So why not have that comment serve as the error message too? Here's an operator, !!; it combines unwrapping with supplying a more descriptive error message to be logged when the application exits:

```
infix operator !!

func !! <T>(wrapped: T?, failureText: @autoclosure () -> String) -> T {
    if let x = wrapped { return x }
    fatalError(failureText())
}
```

Now you can write a more descriptive error message, including the value you expected to be able to unwrap:

```
let s = "foo"
let i = Int(s) !! "Expecting integer, got\"\(s)\""
```

Asserting in Debug Builds

Still, choosing to crash even on release builds is quite a bold move. Often, you might prefer to assert during debug and test builds, but in production, you'd substitute a valid default value — perhaps zero or an empty array.

Enter the interrobang operator, !?. We define this operator to assert on failed unwraps and also to substitute a default value when the assertion doesn't trigger in release mode:

```
infix operator !?

func !?<T: ExpressibleByIntegerLiteral>
    (wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? 0
}
```

Now, the following will assert while debugging but print 0 in release:

```
let s = "20"
let i = Int(s) !? "Expecting integer, got\"\(s)\""
```

Overloading for other literal convertible protocols enables a broad coverage of types that can be defaulted:

```
func !?<T: ExpressibleByArrayLiteral>
    (wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? []
}

func !?<T: ExpressibleByStringLiteral>
    (wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText)
    return wrapped ?? ""
}
```

And for when you want to provide a different explicit default, or for non-standard types, we can define a version that takes a pair — the default and the error text:

```
func !?<T>(wrapped: T?,
    nilDefault: @autoclosure () -> (value: T, text: String)) -> T
{
    assert(wrapped != nil, nilDefault().text)
    return wrapped ?? nilDefault().value
}

// Asserts in debug, returns 5 in release
Int(s) !? (5, "Expected integer")
```

Since optionally chained method calls on methods that return Void return Void?, you can also write a non-generic version to detect when an optional chain hits a nil, resulting in a no-op:

```
func !?(wrapped: ()?, failureText: @autoclosure () -> String) {
    assert(wrapped != nil, failureText)
}

var output: String? = nil
output?.write("something") !? "Wasn't expecting chained nil here"
```

There are three ways to halt execution. The first option, fatalError, takes a message and stops execution unconditionally. The second option, assert, checks a condition and a message and stops execution if the condition evaluates to false. In release builds, the assert gets removed — the condition isn't checked (and execution is never halted). The third option is precondition, which has the same interface as `assert`, but doesn't get removed from release builds, so if the condition evaluates to `false`, execution is stopped.

Living Dangerously: Implicitly Unwrapped Optionals

Make no mistake: implicitly unwrapped optionals — types marked with an exclamation point, such as `UIView!` — are still optionals, albeit ones that are automatically force-unwrapped whenever you use them. Now that we know force-unwraps will crash your application if they're ever `nil`, why on earth would you use them? Well, two reasons really.

Reason 1: Temporarily, because you're calling Objective-C code that hasn't been audited for nullability.

The sole reason implicitly unwrapped optionals even exist is to make interoperability with Objective-C and C easier. Of course, on the first day you start writing Swift against an existing Objective-C code base, any Objective-C method that returns a reference will translate into an implicitly unwrapped optional. Since, for most of Objective-C's lifetime, there was no way to indicate that a reference was nullable, there was little option other than to assume any call returning a reference might return a `nil` reference. But few Objective-C APIs *actually* return null references, so it'd be incredibly annoying to automatically expose them as optionals. Since everyone was used to dealing with the "maybe null" world of Objective-C objects, implicitly unwrapped optionals were a reasonable compromise.

So you see them in unaudited bridged Objective-C code. But you should *never* see a pure native Swift API returning an implicit optional (or passing one into a callback).

Reason 2: Because a value is nil *very* briefly, for a well-defined period of time, and is then never nil again.

For example, if you have a two-phase initialization, then by the time your class is ready to use, the implicitly wrapped optionals will all have a value. This is the reason Xcode/Interface Builder uses them in the view controller lifecycle: in Cocoa and Cocoa Touch, view controllers create their view lazily, so there exists a time window — after a view controller has been initialized but before it has loaded its view — when the view objects its outlets reference have not yet been created.

Implicit Optional Behavior

While implicitly unwrapped optionals usually behave like non-optional values, you can still use most of the unwrap techniques to safely handle them like optionals — chaining, nil-coalescing, if let, map, or just comparing them to nil all work the same:

```
var s: String! = "Hello"
s?.isEmpty // Optional(false)
if let s = s { print(s) }
s = nil
s ?? "Goodbye" // Goodbye
```

As much as implicit optionals try to hide their optional-ness from you, there are a few times when they behave slightly differently than plain values. For example, you can't pass an implicit optional into a function that takes the wrapped type as an inout:

```
func increment(_ x: inout Int) {
    x += 1
}

var i = 1 // Regular Int
```

```
increment(&i) // Increments i to 2
var j: Int! = 1 // Implicitly unwrapped Int
increment(&j)
// Error: Cannot pass immutable value of type 'Int' as inout argument
```

The reason is that inout requires an *lvalue*, which optionals (implicitly unwrapped or not) aren't. We'll have more to say on lvalues in the chapter on functions.

Recap

Optionals are touted as one of Swift's biggest features for writing safer code, and we certainly agree. If you think about it though, the real breakthrough isn't optionals — it's *non-optionals*. Almost every mainstream language has the concept of "null" or "nil"; what most of them don't have is the ability to declare a value as "never nil." Or, alternatively, some types (like non-class types in Objective-C or Java) are "always non-nil," forcing developers to come up with magic values to represent the absence of a value.

APIs whose inputs and outputs are carefully designed with optionals in mind are more expressive and easier to use; there's less need to refer to the documentation because the types carry more information.

All the unwrapping techniques we demonstrated in this chapter are Swift's attempt to make bridging the two worlds of optional and non-optional values as painless as possible. Which method you should use is often a matter of personal preference.

Structs and Classes

In Swift, we can choose from multiple options to store structured data: structs, enums, classes, and capturing variables with closures. Most of the public types in Swift's standard library are defined as structs, with enums and classes making up a much smaller percentage. Part of this may be the nature of the types in the standard library, but it does give an indication as to the importance of structs in Swift. Likewise, many of the classes in Foundation now have struct counterparts specifically built for Swift. That said, we'll focus on the differences between structs and classes in this chapter. Meanwhile, enums behave in a way that's similar to structs.

Here are some of the major things that help distinguish between structs and classes:

- Structs (and enums) are *value types*, whereas classes are *reference types*. When designing with structs, we can ask the compiler to enforce immutability. With classes, we have to enforce it ourselves.

- How memory is managed differs. Structs can be held and accessed directly, whereas class instances are always accessed indirectly through their references. Structs aren't referenced but instead copied. Structs have a single owner, whereas class instances can have many owners.

- With classes, we can use inheritance to share code. With structs (and enums), inheritance isn't possible. Instead, to share code using structs and enums, we need to use different techniques, such as composition, generics, and protocol extensions.

In this chapter, we'll explore these differences in more detail. We'll start by looking at the differences between entities and values. Next, we'll continue by discussing issues with mutability. Then we'll take a look at structs and mutability. After that, we'll demonstrate how to wrap a reference type in a struct in order to use it as an efficient value type. Finally, we'll compare the differences in how memory is managed — particularly how memory is managed in combination with closures, and how to avoid reference cycles.

Value Types

We're often dealing with objects that need an explicit *lifecycle*: they're initialized, changed, and destroyed. For example, a file handle has a clear lifecycle: it's opened, actions are performed on it, and then we need to close it. If we open two file handles that otherwise have the same properties, we still want to keep them separate. In order to compare two file handles, we check whether they point to the same address in memory. Because we compare addresses, file handles are best implemented as reference types, using objects. This is what the `FileHandle` class in Foundation does.

Other types don't need to have a lifecycle. For example, a URL is created and then never changed. More importantly, it doesn't need to perform any action when it's destroyed (in contrast to the file handle, which needs to be closed). When we compare two URL variables, we don't care whether they point to the same address in memory, rather we check whether they point to the same URL. Because we compare URLs by their properties, we say that they're *values*. In Objective-C, they were implemented as immutable objects using the `NSURL` class. However, the Swift counterpart, `URL`, is a struct.

In all software, there are many objects that have a lifecycle — file handles, notification centers, networking interfaces, database connections, and view controllers are some examples. For all these types, we want to perform specific actions on initialization and when they're destroyed. When comparing these types, we don't compare their properties, but instead compare their memory addresses. All of these types are implemented as classes, and all of them are reference types.

There are also many values at play in most software. URLs, binary data, dates, errors, strings, notifications, and numbers are only defined by their properties. When we compare them, we're not interested in their memory addresses. All of these types can be implemented using structs.

Values never change; they're immutable. This is (mostly) a good thing, because code that works with immutable data is much easier to understand. The immutability automatically makes such code thread-safe too: anything that can't change can be safely shared across threads.

In Swift, structs are designed to build values. Structs can't be compared by reference; we can only compare their properties. And although we can declare mutable struct *variables* (using `var`), it's important to understand that the mutability only refers to the variable and not the underlying value. Mutating a property of a struct variable is conceptually the same as assigning a whole new struct (with a different value for the property) to the variable.

Structs have a single owner. For instance, if we pass a struct variable to a function, that function receives a copy of the struct, and it can only change its own copy. This is called *value semantics* (sometimes also called copy semantics). Contrast this with the way objects work: they get passed by reference and can have many owners. This is called *reference semantics*.

Because structs only have a single owner, it's not possible to create a reference cycle. But with classes and functions, we need to always be careful to not create reference cycles. We'll look at reference cycles in the section on memory.

The fact that values are copied all the time may sound inefficient; however, the compiler can optimize away many superfluous copy operations. It can do this because structs are very basic things. A struct copy is a shallow bitwise copy (except if it contains any classes — then it needs to increase the reference count for those). When structs are declared with `let`, the compiler knows for certain that none of those bits can be mutated later on. And there are no hooks for the developer to know *when* the struct is being copied, unlike with similar value types in C++. This simplicity gives the compiler many more possibilities for eliminating copy operations or optimizing a constant structure to be passed by reference rather than by value.

Copy optimizations of a value *type* that might be done by the compiler aren't the same as the copy-on-write behavior of a type with value *semantics*. Copy-on-write has to be implemented by the developer, and it works by detecting that the contained class has shared references.

Unlike the automatic elimination of value type copies, you don't get copy-on-write for free. But the two optimizations — the compiler potentially eliminating unnecessary "dumb" shallow copies, and the code inside types like `Array` that perform "smart" copy-on-write — complement each other. We'll look at how to implement your own copy-on-write mechanism shortly.

If a struct is composed out of other structs, the compiler can enforce immutability. Also, when using structs, the compiler can generate really fast code. For example, the performance of operations on an array containing just structs is usually much better than the performance of operations on an array containing objects. This is because structs usually have less indirection: the values are stored directly inside the array's memory, whereas an array containing objects contains just the references to the objects. Finally, in many cases, the compiler can put structs on the stack rather than on the heap.

When interfacing with Cocoa and Objective-C, we often need to use classes. For example, when implementing a delegate for a table view, there's no choice: we must use a class. Many of Apple's frameworks rely heavily on subclassing. However, depending on the problem domain, we can still create a class where the objects are values. For example, in the Core Image framework, the `CIImage` objects are immutable: they represent an image that never changes.

It's not always easy to decide whether your new type should be a struct or a class. Both behave very differently, and knowing the differences will help you make a decision. In the examples in the rest of this chapter, we'll look in more detail at the implications of value types and provide some guidance for when to use structs.

Mutability

In recent years, manipulating mutable state has earned a bad reputation, and often deservedly so. It's named as a major cause of bugs, and most experts recommend you work with immutable objects where possible, in order to write safe, maintainable code. Luckily, Swift allows us to write safe code while preserving an intuitive mutable coding style at the same time.

To see how this works, let's start by showing some of the problems with mutation. In Foundation, there are two classes for arrays: NSArray, and its subclass, NSMutableArray. We can write the following (crashing) program using NSMutableArray:

```
let mutableArray: NSMutableArray = [1,2,3]
for _ in mutableArray {
    mutableArray.removeLastObject()
}
```

You're not allowed to mutate an NSMutableArray while you're iterating through it, because the iterator works on the original array, and mutating it corrupts the iterator's internal state. Once you know this, the restriction makes sense, and you won't make that mistake again. However, consider that there could be a different method call in the place of mutableArray.removeLastObject(), and that method might mutate mutableArray. Now the violation becomes much harder to see, unless you know exactly what that method does.

Now let's consider the same example, but using Swift arrays:

```
var mutableArray = [1, 2, 3]
for _ in mutableArray {
    mutableArray.removeLast()
}
```

This example doesn't crash, because the iterator keeps a local, independent copy of the array. To see this even more clearly, you could write removeAll instead of removeLast, and if you open it up in a playground, you'll see that the statement gets executed three times because the iterator's copy of the array still contains three elements.

Classes are reference types. If you create an instance of a class and assign it to a new variable, both variables point to the same object:

```
let mutableArray: NSMutableArray = [1, 2, 3]
let otherArray = mutableArray
mutableArray.add(4)
otherArray // ( 1, 2, 3, 4 )
```

Because both variables refer to the same object, they now both refer to the array [1, 2, 3, 4], since changing the value of one variable also changes the value of the other variable. This is a very powerful thing, but it's also a great source of bugs. Calling a method might change something you didn't expect to change, and your invariant won't hold anymore.

Inside a class, we can control mutable and immutable properties with var and let. For example, we could create our own variant of Foundation's Scanner, but for binary data. The Scanner class allows you to scan values from a string, advancing through the string with each successful scanned value. A Scanner does this by storing a string and its current position in the string. Similarly, our class, BinaryScanner, contains the position (which is mutable, hence declared with var) and the original data (which will never change, hence declared with let). This is all we need to store in order to replicate the behavior of Scanner:

```
class BinaryScanner {
    var position: Int
    let data: Data
    init(data: Data) {
        self.position = 0
        self.data = data
    }
}
```

We can also add a method that scans a byte. Note that this method is mutating: it changes the position (unless we've reached the end of the data):

```
extension BinaryScanner {
    func scanByte() -> UInt8? {
        guard position < data.endIndex else {
            return nil
        }
        position += 1
        return data[position-1]
    }
}
```

To test it out, we can write a function that scans all the remaining bytes and prints them:

```
func scanRemainingBytes(scanner: BinaryScanner) {
    while let byte = scanner.scanByte() {
        print(byte)
    }
}
let scanner = BinaryScanner(data: Data("hi".utf8))
scanRemainingBytes(scanner: scanner)
/*
104
105
*/
```

Everything works as expected. However, it's easy to construct an example with a race condition. If we use Grand Central Dispatch to call scanRemainingBytes from two different threads, it'll eventually run into a race condition. In the code below, the condition position < data.endIndex can be true on one thread, but then GCD switches to another thread and scans the last byte. Now, when control switches back to the first thread, the position will be incremented and the subscript will access a value that's out of bounds:

```
for _ in 0..<Int.max {
    let newScanner = BinaryScanner(data: Data("hi".utf8))
    DispatchQueue.global().async {
        scanRemainingBytes(scanner: newScanner)
    }
    scanRemainingBytes(scanner: newScanner)
}
```

The race condition doesn't occur too often (hence the Int.max), and is therefore difficult to find during testing. If we change BinaryScanner to a struct, this problem doesn't occur at all. In the next section, we'll look at why.

Structs

Value types imply that whenever a variable is assigned to another variable, the value itself — and not just a reference to the value — is copied. For example, in almost all programming languages, scalar types are value types. This means that whenever a value is assigned to a new variable, it's copied rather than passed by reference:

```
var a = 42
var b = a
b += 1
b // 43
a // 42
```

After the above code executes, the value of b will be 43, but a will still be 42. This is so natural that it seems like stating the obvious. However, in Swift, all structs behave this way — not just scalar types.

Let's start with a simple struct that describes a Point. This is similar to CGPoint, except that it contains Ints, whereas CGPoint contains CGFloats:

```
struct Point {
    var x: Int
    var y: Int
}
```

For structs, Swift automatically adds a memberwise initializer. This means we can now initialize a new variable:

```
let origin = Point(x: 0, y: 0)
```

Because structs in Swift have value semantics, we can't change any of the properties of a struct variable that's defined using let. For example, the following code won't work:

```
origin.x = 10 // Error
```

Even though we defined x within the struct as a var property, we can't change it, because origin is declared using let. This has some major advantages. For example, if you read a line like let point = ..., and you know that point is a struct variable with value semantics, then you also know that it'll never, ever, change. This is incredibly helpful when reading through code.

To create a mutable variable, we need to use var:

```
var otherPoint = Point(x: 0, y: 0)
otherPoint.x += 10
otherPoint // (x: 10, y: 0)
```

Unlike with objects, every struct variable is unique. For example, we can create a new variable, thirdPoint, and assign the value of origin to it. Now we can change thirdPoint, but origin (which we defined as an immutable variable using let) won't change:

```
var thirdPoint = origin
thirdPoint.x += 10
thirdPoint // (x: 10, y: 0)
origin // (x: 0, y: 0)
```

When you assign a struct to a new variable, Swift automatically makes a copy. Even though this sounds very expensive, many of the copies can be optimized away by the compiler, and Swift tries hard to make the copies very cheap. In fact, many structs in the standard library are implemented using a technique called copy-on-write, which we'll look at later.

Structs can also contain other structs. For example, if we define a Size struct, we can create a Rectangle struct, which is composed out of a point and a size:

```
struct Size {
    var width: Int
    var height: Int
}
```

```
struct Rectangle {
    var origin: Point
    var size: Size
}
```

Just like before, we get a memberwise initializer for Rectangle. Here, we first define a *type property* on Point, i.e. a property that exists on the type rather than on instances of the type. The type property Point.zero effectively acts like a factory initializer for a commonly used specific Point instance. We then use that value to initialize a Rectangle:

```
extension Point {
    static let zero = Point(x: 0, y: 0)
}

let rect = Rectangle(origin: Point.zero,
    size: Size(width: 320, height: 480))
```

If we want a custom initializer for our struct, we can add it directly inside the struct definition. However, if the struct definition contains a custom initializer, Swift doesn't generate a memberwise initializer. By defining our custom initializer in an extension, we also get to keep the memberwise initializer:

```
extension Rectangle {
    init(x: Int = 0, y: Int = 0, width: Int, height: Int) {
        origin = Point(x: x, y: y)
        size = Size(width: width, height: height)
    }
}
```

Instead of setting origin and size directly, we could've also called self.init(origin:size:).

Mutation Semantics

If we define a mutable variable screen, we can add a didSet block that gets executed whenever screen is changed. This didSet works for every definition of a struct, be it in a playground, as a member of a class or other struct, or as a global variable:

```
var screen = Rectangle(width: 320, height: 480) {
    didSet {
        print("Screen changed: \(screen)")
    }
}
```

Maybe somewhat surprisingly, even if we change something deep inside the struct, the didSet handler will get triggered:

```
screen.origin.x = 10 // Screen changed: (10, 0, 320, 480)
```

Understanding why this works is key to understanding value types. Mutating a struct variable is semantically the same as assigning a new value to it. And even if only one property of a larger struct gets mutated, it's the equivalent of replacing the entire struct with a new value. A mutation deep inside a tree of nested structs trickles all the way up the tree to the outermost instance, triggering any willSet and didSet handlers it encounters along the way.

Although, semantically, the entire struct is replaced with a new one, this is usually no less efficient; the compiler can still mutate the value in place. Since the struct has no other owner, it doesn't actually need to make a copy. With copy-on-write structs (which we'll discuss later), this works differently.

Since the collection types in the standard library are structs, they naturally behave the same way. Appending an item to an array will trigger the array's didSet handler, as will mutating one of the array's elements directly through a subscript:

```
var screens: [Rectangle] = [] {
    didSet {
        print("Screens array changed: \(screens)")
    }
}
screens.append(Rectangle(width: 320, height: 480))
// Screens array changed: [(0, 0, 320, 480)]
screens[0].origin.x += 100 // Screens array changed: [(100, 0, 320, 480)]
```

The didSet trigger wouldn't fire if Rectangle were a class, because in that case, the reference the array stores doesn't change — only the object it's referring to does.

Mutating Methods

Assume we want to add a translate method to Rectangle, which moves the rectangle by a given offset. We'll need to add the offset to the rectangle's current origin, so let's start by adding an overload for the + operator that adds two Points together and returns a new Point:

```
func +(lhs: Point, rhs: Point) -> Point {
    return Point(x: lhs.x + rhs.x, y: lhs.y + rhs.y)
}
screen.origin + Point(x: 10, y: 10) // (x: 10, y: 10)
```

We now have everything we need to implement translate, yet our first attempt doesn't work:

```
extension Rectangle {
    func translate(by offset: Point) {
        // Error: Cannot assign to property: 'self' is immutable
        origin = origin + offset
    }
}
```

The compiler complains that we can't assign to the origin property because self is immutable (writing origin = is shorthand for self.origin =). We can think of self as an extra, implicit parameter that gets passed to each method. You never have to pass the parameter, but it's always there inside the method body, and it's immutable so that value semantics can be guaranteed. If we want to mutate self, or any property of self, or even nested properties (e.g. self.origin.x), we need to mark the method as mutating:

```
extension Rectangle {
    mutating func translate(by offset: Point) {
        origin = origin + offset
    }
}
```

```
screen.translate(by: Point(x: 10, y: 10))
screen // (10, 10, 320, 480)
```

The compiler enforces the mutating keyword. Unless we use it, we're not allowed to mutate any part of self inside the method. By marking the method as mutating, we change the behavior of self inside that method. Instead of it being a let, it now works like a var: we can freely change any of its mutable properties. (To be precise, it's not even a var, but we'll get to that in a little bit).

The mutating keyword also determines which methods can be called on variables declared with let. Anything that's marked as mutating may only be called on instances declared with var:

```
let otherScreen = screen
// Error: Cannot use mutating member on immutable value
otherScreen.translate(by: Point(x: 10, y: 10))
```

Thinking back to the built-in collections chapter, we can now see how the difference between let and var applies to collections as well. The append method on arrays is defined as mutating, and therefore, we're not allowed to call it on an array declared with let.

Property setters are implicitly mutating; that's why you can't call a setter on a let variable:

```
let point = Point.zero
// Error: Cannot assign to property: 'point' is a 'let' constant
point.x = 10
```

> *mutating is also how willSet and didSet "know" when to fire: any invocation of a mutating method or an implicitly mutating setter will trigger the observers.*

In many cases, it makes sense to have both a mutable and an immutable variant of the same method. For example, arrays have both a sort() method (which is mutating and sorts in place) and a sorted() method (which returns a new array). We can also add a non-mutating variant of our translate(by:_) method. Instead of mutating self, we create a copy, mutate that, and return a new Rectangle:

```
extension Rectangle {
    func translated(by offset: Point) -> Rectangle {
        var copy = self
        copy.translate(by: offset)
        return copy
    }
}
screen.translated(by: Point(x: 20, y: 20)) // (30, 30, 320, 480)
```

The names sort and sorted aren't chosen at random — they conform to the Swift [API Design Guidelines](). Methods that have a side effect should read as an imperative verb phrase, such as sort. Non-mutating variants of those methods should have an -ed or -ing suffix. We applied the same guidelines to translate and translated.

As we saw earlier, it's easy to introduce bugs when working with mutable objects. Swift structs with mutable properties and mutating methods don't have this problem. The mutation of the struct is a local side effect, and it only applies to the variable being modified. Because every struct variable is unique (or in other words: every struct value has exactly one owner), it's almost impossible to introduce bugs this way — that is, unless you're accessing a global or captured struct variable from multiple threads (closures capture by reference by default).

How mutating Works: inout Parameters

To understand how the mutating keyword works, we need to take a look at the inout keyword. Before we do that, let's define a free function that moves a rectangle by 10 points on both axes. We can't simply call translate directly on the rectangle parameter, because all function parameters are immutable by default and get passed in as copies. Instead, we need to use translated(by:) and return the translated rectangle as a new value. Callers that want to use the function to mutate an existing value have to reassign the result manually:

```
func translatedByTenTen(rectangle: Rectangle) -> Rectangle {
    return rectangle.translated(by: Point(x: 10, y: 10))
}
```

```
screen = translatedByTenTen(rectangle: screen)
screen // (10, 10, 320, 480)
```

How could we write a function (not a method) that changes a Rectangle in place? We've seen that the mutating keyword does exactly that for methods: it makes the implicit self parameter mutable.

Free functions can achieve the same effect by marking one or more parameters as inout. Just like with a regular parameter, a copy of the value gets passed in to the function. However, inside the function body the argument becomes mutable — it's as if it were defined as a var. And once the function returns, Swift copies the (possibly mutated) value from the function back to the caller, overwriting the original value.

Using inout, we can write a translate function that mutates the rectangle that gets passed in. Notice that we're now allowed to use translate(by:) in the function body because rectangle is mutable:

```
func translateByTwentyTwenty(rectangle: inout Rectangle) {
    rectangle.translate(by: Point(x: 20, y: 20))
}
translateByTwentyTwenty(rectangle: &screen)
screen // (30, 30, 320, 480)
```

The translateByTwentyTwenty function takes the screen rectangle, changes it locally, and copies the new value back (overriding the previous value of screen). This behavior is exactly the same as that of a mutating method. In fact, mutating methods are just like regular methods on the struct, except the implicit self parameter is marked as inout.

Accordingly, we can't call translateByTwentyTwenty on a rectangle that's defined using let. We can only use it with mutable values:

```
let immutableScreen = screen
// Error: Cannot pass immutable value as inout argument
translateByTwentyTwenty(rectangle: &immutableScreen)
```

We'll go into more detail about inout in the functions chapter. For now, it suffices to say that inout is in lots of places. For example, it's now easy to understand how mutating a value through a subscript works:

```
var array = [Point(x: 0, y: 0), Point(x: 10, y: 10)]
array[0] += Point(x: 100, y: 100)
array // [(x: 100, y: 100), (x: 10, y: 10)]
```

The expression array[0] is automatically passed in as an inout variable. In the functions chapter, we'll see why we can use some expressions (like array[0]) as inout parameters and not others.

Operators that mutate their left-hand side, such as +=, also require that parameter to be inout. Here's an implementation of += for Point. The compiler wouldn't allow the assignment to lhs if we omitted the inout:

```
func +=(lhs: inout Point, rhs: Point) {
    lhs = lhs + rhs
}

var myPoint = Point.zero
myPoint += Point(x: 10, y: 10)
myPoint // (x: 10, y: 10)
```

Value Types Can Help Avoid Concurrency Bugs

Let's revisit the BinaryScanner example from the introduction of this chapter. We had the following problematic code snippet:

```
for _ in 0..<Int.max {
    let newScanner = BinaryScanner(data: Data("hi".utf8))
    DispatchQueue.global().async {
        scanRemainingBytes(scanner: newScanner)
    }
    scanRemainingBytes(scanner: newScanner)
}
```

If we make BinaryScanner a struct instead of a class, each call to scanRemainingBytes gets its own independent copy of newScanner. Therefore, the calls can safely keep iterating over the array without having to worry that the struct gets mutated from a different method or thread. Since the two threads no longer share a single position value, this small change fundamentally alters the program's behavior: it'll now print each byte twice per loop iteration.

Also keep in mind that structs don't magically make your code thread-safe. For example, if we keep BinaryScanner as a struct but inline the scanRemainingBytes method, we end up with the same race condition we had before. The two while loops refer to the same newScanner variable and will mutate it from different threads at the same time:

```
for _ in 0..<Int.max {
    let newScanner = BinaryScanner(data: Data("hi".utf8))
    DispatchQueue.global().async {
        while let byte = newScanner.scanByte() {
            print(byte)
        }
    }
    while let byte = newScanner.scanByte() {
        print(byte)
    }
}
```

Copy-On-Write

In the Swift standard library, collections like Array, Dictionary, and Set are implemented using a technique called *copy-on-write*. Let's say we have an array of integers:

```
var x = [1,2,3]
```

If we create a new variable, y, and assign the value of x to it, a copy gets made, and both x and y now contain independent structs:

```
var y = x
```

Internally, each of these Array structs contains a reference to some memory buffer. This buffer is where the actual elements of the array are stored. At this point, both arrays reference the same buffer — the arrays are sharing this part of their storage. However, the moment we mutate x, the shared reference gets detected, and the buffer is copied.

This means we can mutate both variables independently, yet the (expensive) copy of the elements only happens when it has to — once we mutate one of the variables:

```
x.append(5)
y.removeLast()
x // [1, 2, 3, 5]
y // [1, 2]
```

This behavior is called *copy-on-write*. The way it works is that whenever an array is mutated, it first checks if its reference to the storage buffer is *unique*, i.e. if the array is the sole owner of the buffer. If so, the buffer can be mutated in place; no copy has to be made. However, if the buffer has more than one owner (as in this example), the array will create a copy of the buffer and then mutate the copy, leaving the other owners unaffected.

As the author of a struct, copy-on-write behavior isn't something you get for free; you have to implement it yourself. Implementing copy-on-write for your own types makes sense whenever you define a struct that contains a mutable reference but should still retain value semantics. Maintaining value semantics would normally require an expensive copy on every mutation, but copy-on-write avoids copying unless it's absolutely necessary.

Implementing Copy-On-Write

As an example of a custom type with copy-on-write behavior, let's recreate the Data struct from Foundation and use the NSMutableData class as our internal reference type. Data has value semantics, and it behaves just like Array:

```
var input: [UInt8] = [0x0b,0xad,0xf0,0x0d]
var other: [UInt8] = [0x0d]
var d = Data(bytes: input)
var e = d
d.append(contentsOf: other)
d // 5 bytes
e // 4 bytes
```

As we can see above, d and e are independent: adding a byte to d doesn't change the value of e.

Writing the same example using NSMutableData, we can see that types with reference semantics behave differently:

```
var f = NSMutableData(bytes: &input, length: input.count)
var g = f
f.append(&other, length: other.count)
f // <0badf00d 0d>
g // <0badf00d 0d>
```

Both f and g refer to the same object (in other words: they point to the same piece of memory), so changing one also changes the other. We can even verify that they refer to the same object by using the === operator:

```
f === g // true
```

If we naively wrap NSMutableData in a struct, we don't get value semantics automatically. For example, we could try the following:

```
struct MyData {
    var _data: NSMutableData
    init(_ data: NSData) {
        _data = data.mutableCopy() as! NSMutableData
    }
}
```

If we copy a struct variable, a shallow copy is made. This means the reference to the NSMutableData object, and not the object itself, will get copied:

```
let theData = NSData(base64Encoded: "wAEP/w==")!
let x = MyData(theData)
let y = x
x._data === y._data // true
```

We can add an append function, which delegates to the underlying _data property, and again, we can see that we've created a struct without value semantics:

```
extension MyData {
    func append(_ byte: UInt8) {
        var mutableByte = byte
        _data.append(&mutableByte, length: 1)
    }
}

x.append(0x55)
y // <c0010fff 55>
```

Because we're only modifying the object _data is referring to, we don't even have to mark append as mutating. After all, the reference stays constant, and the struct too. Therefore, we were able to declare x and y using let, even though they were mutable.

Copy-On-Write (The Expensive Way)

To arrive at a working implementation, let's take the term *copy-on-write* literally: every time we need to mutate _data, we'll make a copy of it first and then mutate the copy. This approach won't be very efficient because we'll make a ton of unnecessary copies, but it will achieve our goal of giving MyData value semantics — other instances referencing the original _data object won't be affected by the mutation.

Instead of mutating _data directly, we access it exclusively through a computed property, _dataForWriting. It's the getter of this property that copies the NSMutableData instance and returns the copy:

```
struct MyData {
    fileprivate var _data: NSMutableData
    fileprivate var _dataForWriting: NSMutableData {
        mutating get {
            _data = _data.mutableCopy() as! NSMutableData
            return _data
        }
    }
    init() {
        _data = NSMutableData()
    }
    init(_ data: NSData) {
        _data = data.mutableCopy() as! NSMutableData
    }
}
```

Because _dataForWriting mutates the struct (it assigns a new value to the _data property), the getter has to be marked as mutating. This means we can only use it on variables declared with var.

We can use _dataForWriting in our append method, which now also needs to be marked as mutating:

```
extension MyData {
    mutating func append(_ byte: UInt8) {
        var mutableByte = byte
        _dataForWriting.append(&mutableByte, length: 1)
    }
}
```

Our struct now has value semantics. If we assign the value of x to the variable y, both variables are still pointing to the same underlying NSMutableData object. However, the moment we use append on either one of the variables, a copy gets made:

```
let theData = NSData(base64Encoded: "wAEP/w==")!
var x = MyData(theData)
let y = x
x._data === y._data // true
x.append(0x55)
y // <c0010fff>
x._data === y._data // false
```

This strategy works, but it’s wasteful if we mutate the same variable multiple times. Consider the following example:

```
var buffer = MyData(NSData())
for byte in 0..<5 as CountableRange<UInt8> {
    buffer.append(byte)
}
```

Each time we call append, the underlying _data object gets copied. Because buffer doesn’t share its storage with other MyData instances, it’d have been a lot more efficient (and just as safe) to mutate it in place.

Copy-On-Write (The Efficient Way)

To provide efficient copy-on-write behavior, we need to know whether an object (e.g. the NSMutableData instance) is uniquely referenced. If it's a unique reference, we can modify the object in place. Otherwise, we create a copy of the object before modifying it. In Swift, we can use the isKnownUniquelyReferenced function to find out if a reference has only one owner. If you pass in an instance of a Swift class to this function, and if no one else has a strong reference to the object, the function returns true. If there are other strong references, it returns false. Unfortunately, it also returns false for Objective-C classes. Therefore, it doesn't make sense to use the function with NSMutableData directly. However, we can write a simple Swift class that wraps any Objective-C object (or any other value) into a Swift object:

```
final class Box<A> {
    var unbox: A
    init(_ value: A) { self.unbox = value }
}

var x = Box(NSMutableData())
isKnownUniquelyReferenced(&x) // true
```

If we have multiple references to the same object, the function will return false:

```
var y = x
isKnownUniquelyReferenced(&x) // false
```

This also works when the references are inside a struct and not just global variables. Using this knowledge, we can now write a variant of MyData, which checks whether _data is uniquely referenced before mutating it. Let's also add a print statement to quickly see during debugging how often we make a copy:

```
struct MyData {
    private var _data: Box<NSMutableData>
    var _dataForWriting: NSMutableData {
        mutating get {
            if !isKnownUniquelyReferenced(&_data) {
                _data = Box(_data.unbox.mutableCopy() as! NSMutableData)
                print("Making a copy")
            }
            return _data.unbox
        }
    }
    init() {
        _data = Box(NSMutableData())
    }
    init(_ data: NSData) {
        _data = Box(data.mutableCopy() as! NSMutableData)
    }
}
```

In the _dataForWriting getter, we now need to unbox the _data property to access the NSMutableData instance. The append method can remain unchanged because the logic for when to make a copy is fully contained inside _dataForWriting:

```
extension MyData {
    mutating func append(_ byte: UInt8) {
        var mutableByte = byte
        _dataForWriting.append(&mutableByte, length: 1)
    }
}
```

To test out our code, let's write the loop again:

```
var bytes = MyData()
var copy = bytes
for byte in 0..<5 as CountableRange<UInt8> {
    print("Appending 0x\(String(byte, radix: 16))")
    bytes.append(byte)
}
/*
Appending 0x0
Making a copy
Appending 0x1
Appending 0x2
Appending 0x3
Appending 0x4
*/
bytes // <00010203 04>
copy // <>
```

If we run the code above, we can see that our debug statement only gets printed once: when we call `append` for the first time. In subsequent iterations, the uniqueness is detected and no copy gets made.

This technique allows you to create custom structs that retain value semantics while still being just as efficient as you would be working with objects or pointers. As a user of the struct, you don't need to worry about copying these structs manually — the implementation will take care of it for you. Because copy-on-write is combined with optimizations done by the compiler, many unnecessary copy operations can be removed.

When you define your own structs and classes, it's important to pay attention to the expected copying and mutability behavior. Structs are expected to have value semantics. When using a class inside a struct, we need to make sure that it's truly immutable. If this isn't possible, we either need to take extra steps (like above), or just use a class, in which case consumers of our data don't expect it to behave like a value.

Most data structures in the Swift standard library are value types using copy-on-write. For example, arrays, dictionaries, sets, and strings are all structs. This makes it simpler to understand code that uses those types. When we pass an array to a function, we know the function can't modify the original array: it only works on a copy of the array. Also, the way arrays are implemented, we know that no unnecessary copies will be made. Contrast this with the Foundation data types, where it's best practice to always manually copy types like `NSArray` and `NSString`. When working with Foundation data types, it's easy to forget to manually copy the object and instead accidentally write unsafe code.

Even when you could create a struct, there might still be very good reasons to use a class. For example, you might want an immutable type that only has a single instance, or maybe you're wrapping a reference type and don't want to implement copy-on-write. Or, you might need to interface with Objective-C, in which case, structs might not work either. By defining a restrictive interface for your class, it's still very possible to make it immutable.

It might also be interesting to wrap existing types in enums, which are also value types, just like structs. In the chapter on wrapping CommonMark, we'll provide an enum-based interface to a reference type.

Copy-On-Write Gotchas

Unfortunately, it's all too easy to introduce accidental copies. To see this behavior, we'll create a very simple struct that contains a reference to an empty Swift class. The struct has a single mutating method, change, which checks whether or not the reference should be copied. Rather than copying, it just returns a string indicating whether or not a copy would have occurred:

```
final class Empty { }

struct COWStruct {
    var ref = Empty()

    mutating func change() -> String {
        if isKnownUniquelyReferenced(&ref) {
            return "No copy"
        } else {
            return "Copy"
        }
        // Perform actual change
    }
}
```

For example, if we create a variable and immediately mutate it, the variable isn't shared. Therefore, the reference is unique and no copy is necessary:

```
var s = COWStruct()
s.change() // No copy
```

As we'd expect, when we create a second variable, the reference is shared, and a copy is necessary the moment we call change:

```
var original = COWStruct()
var copy = original
original.change() // Copy
```

When we put one of our structs in an array, we can mutate the array element directly, and no copy is necessary. This is because accessing an element via array subscripting has direct access to the memory location:

```
var array = [COWStruct()]
array[0].change() // No copy
```

If we go through an intermediate variable, though, a copy is made:

```
var otherArray = [COWStruct()]
var x = array[0]
x.change() // Copy
```

Somewhat surprisingly, all other types — including dictionaries, sets, and your own types — behave very differently. For example, the dictionary subscript looks up the value in the dictionary and then returns the value. Because we're dealing with value semantics, a copy is returned. Therefore, calling `change()` on that value will make a copy, as the `COWStruct` is no longer uniquely referenced:

```
var dict = ["key": COWStruct()]
dict["key"]?.change() // Optional("Copy")
```

If you want to avoid making copies when you put a copy-on-write struct inside a dictionary, you can wrap the value inside a class box, effectively giving the value reference semantics.

When you're working with your own structs, you need to keep this in mind. For example, if we create a container that simply stores a value, we can either modify the storage property directly, or we can access it indirectly, such as through a subscript. When we directly access it, we get the copy-on-write optimization, but when we access it indirectly through a subscript, a copy is made:

```
struct ContainerStruct<A> {
    var storage: A
    subscript(s: String) -> A {
        get { return storage }
        set { storage = newValue }
    }
}

var d = ContainerStruct(storage: COWStruct())
d.storage.change() // No copy
d["test"].change() // Copy
```

The implementation of the Array subscript uses a special technique to make copy-on-write work, but unfortunately, no other types currently use this. The Swift team mentioned that they hope to generalize it to dictionaries.

The way Array implements the subscript is by using *addressors*. An addressor allows for direct access to memory. Instead of returning the element, an array subscript returns an addressor for the element. This way, the element memory can be modified in place, and unnecessary copies can be eliminated. You can use addressors in your own code, but as they're not officially documented, they're bound to change. For more information, see the Accessors.rst document in the Swift repository.

Closures and Mutability

In this section, we'll look at how closures store data.

For example, consider a function that generates a unique integer every time it gets called (until it reaches Int.max). It works by moving the state outside of the function. In other words, it *closes* over the variable i:

```
var i = 0
func uniqueInteger() -> Int {
    i += 1
    return i
}
```

Every time we call this function, the shared variable i will change, and a different integer will be returned. Functions are reference types as well — if we assign uniqueInteger to another variable, the compiler won't copy the function (or i). Instead, it'll create a reference to the same function:

```
let otherFunction: () -> Int = uniqueInteger
```

Calling otherFunction will have exactly the same effect as calling uniqueInteger. This is true for all closures and functions: if we pass them around, they always get passed by reference, and they always share the same state.

Recall the function-based fibsIterator example from the collection protocols chapter, where we saw this behavior before. When we used the iterator, the iterator itself (being a function) was mutating its state. In order to create a fresh iterator for each iteration, we had to wrap it in an AnySequence.

If we want to have multiple different unique integer providers, we can use the same technique: instead of returning the integer, we return a closure that captures the mutable variable. The returned closure is a reference type, and passing it around will share the state. However, calling uniqueIntegerProvider repeatedly returns a fresh function that starts at zero every time:

```
func uniqueIntegerProvider() -> () -> Int {
    var i = 0
    return {
        i += 1
        return i
    }
}
```

Instead of returning a closure, we can also wrap the behavior in an AnyIterator. That way, we can even use our integer provider in a for loop:

```
func uniqueIntegerProvider() -> AnyIterator<Int> {
    var i = 0
    return AnyIterator {
        i += 1
        return i
    }
}
```

Swift structs are commonly stored on the stack rather than on the heap. For mutable structs this is an optimization, though; a struct variable is created on the heap by default, and the optimizer will then move it to the stack in almost all cases. The reason the compiler works in this way is that variables that are captured by an escaping closure need to outlive their stack frame. When the compiler recognizes that a struct variable is closed over by a function, it doesn't apply the optimization and the struct remains on the heap. That way, the `i` variable in our example persists even when the scope of `uniqueIntegerProvider` exits.

Memory

Value types are very common in Swift, and memory management for them is very easy. Because they have a single owner, the memory needed for them is created and freed automatically. When using value types, you can't create cyclic references. For example, consider the following snippet:

```
struct Person {
    let name: String
    var parents: [Person]
}
var john = Person(name: "John", parents: [])
john.parents = [john]
john // John, parents: [John, parents: []]
```

Because of the way value types work, the moment we put `john` in an array, a copy is created. It'd be more precise to say: "we put the value of `john` in an array." If `Person` were a class, we'd now have a cycle. With the struct version, `john` now has a single parent, which is the initial value of `john` with an empty parents array.

For classes, Swift uses automatic reference counting (ARC) to manage memory. In most cases, this means things will work as expected. Every time you create a new reference to an object (for example, when you assign a value to a class variable), the reference count gets increased by one. Once you let go of that reference (for example, the variable goes out of scope), the reference count decreases by one. When the reference count goes to zero, the object is deallocated. A variable that behaves in this manner is also called a *strong reference* (compared to weak or unowned references, which we'll discuss in a bit).

For example, consider the following code:

```
class View {
    var window: Window
    init(window: Window) {
        self.window = window
    }
}

class Window {
    var rootView: View?
}
```

We can now allocate and initialize a window and assign the instance to a variable. After the first line, the reference count is one. The moment we set the variable to nil, the reference count of our Window instance is zero, and the instance gets deallocated:

```
var myWindow: Window? = Window() // refcount: 1
myWindow = nil // refcount: 0, deallocating
```

When comparing Swift to a garbage-collected language, at first glance it looks like things are very similar when it comes to memory management. Most times, you don't even think about it. However, consider the following example:

```
var window: Window? = Window() // window: 1
var view: View? = View(window: window!) // window: 2, view: 1
window?.rootView = view // window: 2, view: 2
view = nil // window: 2, view: 1
window = nil // window: 1, view: 1
```

First, the window gets created, and the reference count for the window will be one. The view gets created and holds a strong reference to the window, so the window's reference count will be two, and the view's reference count will be one. Then, assigning the view as the window's rootView will increase the view's reference count by one. Now, both the view and the window have a reference count of two. After setting both variables to nil, they still have a reference count of one. Even though they're no longer accessible from a variable, they strongly reference each other. This is called a *reference cycle*, and when dealing with graph-like data structures, we need to be very aware of this. Because of the reference cycle, these two objects will never be deallocated during the lifetime of the program.

Weak References

To break the reference cycle, we need to make sure that one of the references is either weak or unowned. When you mark a variable as weak, assigning a value to it doesn't change the reference count. Weak references in Swift are always *zeroing*: the variable will be automatically set to nil once the referred object gets deallocated — this is why weak references must always be optionals.

Rewriting the example above, we could make the window's rootView property weak, which means it won't strongly reference the view and automatically becomes nil once the view is deallocated. To see what's going on, we can add some print statements to the classes' deinitializers. deinit gets called just before a class deallocates:

```swift
class View {
    var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}

class Window {
    weak var rootView: View?
    deinit {
        print("Deinit Window")
    }
}
```

In the code below, we again create a window and a view. As before, the view strongly references the window; but because the window's rootView is declared as weak, the window doesn't strongly reference the view anymore. This way, we have no reference cycle, and both objects get deallocated when we set the variables to nil:

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
window = nil
view = nil
/*
Deinit View
Deinit Window
*/
```

Weak references are very useful when working with delegates, as is common in Cocoa. The delegating object (e.g. a table view) needs a reference to its delegate, but it shouldn't *own* the delegate because that would likely create a reference cycle. Therefore, delegate references are usually weak, and another object (e.g. a view controller) is responsible for making sure the delegate stays around for as long as needed.

Unowned References

Weak references must always be optional types because they can become nil, but sometimes we may not want this. For example, maybe we know that our views will always have a window (so the property shouldn't be optional), but we don't want a view to strongly reference the window. In other words, assigning to the property should leave the reference count unchanged. For these situations, there's the unowned keyword:

```
class View {
    unowned var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}

class Window {
    var rootView: View?
    deinit {
        print("Deinit Window")
    }
}
```

In the code below, we can see that both objects get deallocated, as in the previous example with a weak reference:

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
view = nil
window = nil
/*
Deinit Window
Deinit View
*/
```

So there's still no reference cycle, but now we're responsible for ensuring that the window outlives the view. If the window is deallocated and the unowned variable is accessed, there'll be a runtime crash.

The Swift runtime keeps a second reference count in the object to keep track of unowned references. When all strong references are gone, the object will release all of its resources (for example, any references to other objects). However, the memory of the object itself will still be there until all unowned references are gone too. The memory is marked as invalid (sometimes also called *zombie* memory), and any time we try to access an unowned reference, a runtime error will occur.

Note that this isn't the same as undefined behavior. There's a third option, `unowned(unsafe)`, which doesn't have this runtime check. If we access an invalid reference that's marked as `unowned(unsafe)`, we get undefined behavior.

Choosing Between Unowned and Weak References

Should you prefer unowned or weak references in your own APIs? Ultimately, this question boils down to the lifetimes of the objects involved. If the objects have *independent lifetimes* — that is, if you can't make any assumptions about which object will outlive the other — a weak reference is the only possible choice.

On the other hand, if you can guarantee that the non-strongly referenced object has *the same lifetime* as its counterpart or will always outlive it, an unowned reference is often more convenient. This is because it doesn't have to be optional and the variable can be declared with `let`, whereas weak references must always be optional `var`s. Same-lifetime situations are very common, especially when the two objects have a parent-child relationship. When the parent controls the child's lifetime with a strong reference and you can guarantee that no other objects know about the child, the child's back reference to its parent can always be unowned.

Unowned references also have less overhead than weak references, so accessing a property or calling a method on an unowned reference will be slightly faster; this should only be a factor in very performance-critical code paths, however.

The downside of preferring unowned references is of course that your program may crash if you make a mistake in your lifetime assumptions. Personally, we often find ourselves preferring `weak` even when `unowned` could be used because the former forces us to explicitly check if the reference is still valid at every point of use. We might want to refactor some code at a later point, and our earlier assumptions about object lifetimes might not be valid anymore.

But there's definitely an argument to be made to always use the modifier that captures the lifetime characteristics you expect your code to have in order to make them explicit. If you or someone else later changes the code in a way that invalidates those assumptions, a hard crash is arguably the sensible way to alert you to the problem — assuming you find the bug during testing.

Closures and Memory

Classes aren't the only kind of reference type in Swift. Functions are reference types too, and this includes closures. As we saw in the section on closures and mutability, a closure can capture variables. If these variables are themselves reference types, the closure will maintain a strong reference to them. For example, if you have a variable handle that's a FileHandle object and you access it within a callback, the callback will increment the reference count for that handle:

```
let handle = FileHandle(forWritingAtPath: "out.html")
let request = URLRequest(url: URL(string: "https://www.objc.io")!)
URLSession.shared.dataTask(with: request) { (data, _, _) in
    guard let theData = data else { return }
    // Closure keeps strong reference to handle
    handle?.write(theData)
}.resume()
```

Once the callback is done, the URL session releases the closure, and the variables it closes over (in the example above, just handle) will have their reference counts decremented. This strong reference to the closed-over variables is necessary, otherwise the variable could already be deallocated when you access it within the callback.

Only closures that can *escape* need to keep strong references to their variables. In the chapter on functions, we'll look into more detail at escaping vs. non-escaping functions.

Reference Cycles with Closures

One of the issues with closures capturing their variables is the (accidental) introduction of reference cycles. The usual pattern is like this: object A references object B, but object B stores a callback that references object A. Let's consider our example from before, where a view references its window, and the window has a weak reference back to its root view. Additionally, the window now has an onRotate callback, which is optional and has an initial value of nil:

```
class View {
    var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}

class Window {
    weak var rootView: View?
    var onRotate: (() -> ())?
    deinit {
        print("Deinit Window")
    }
}
```

If we create a view and set up the window like before, all is well, and we don't have a reference cycle yet:

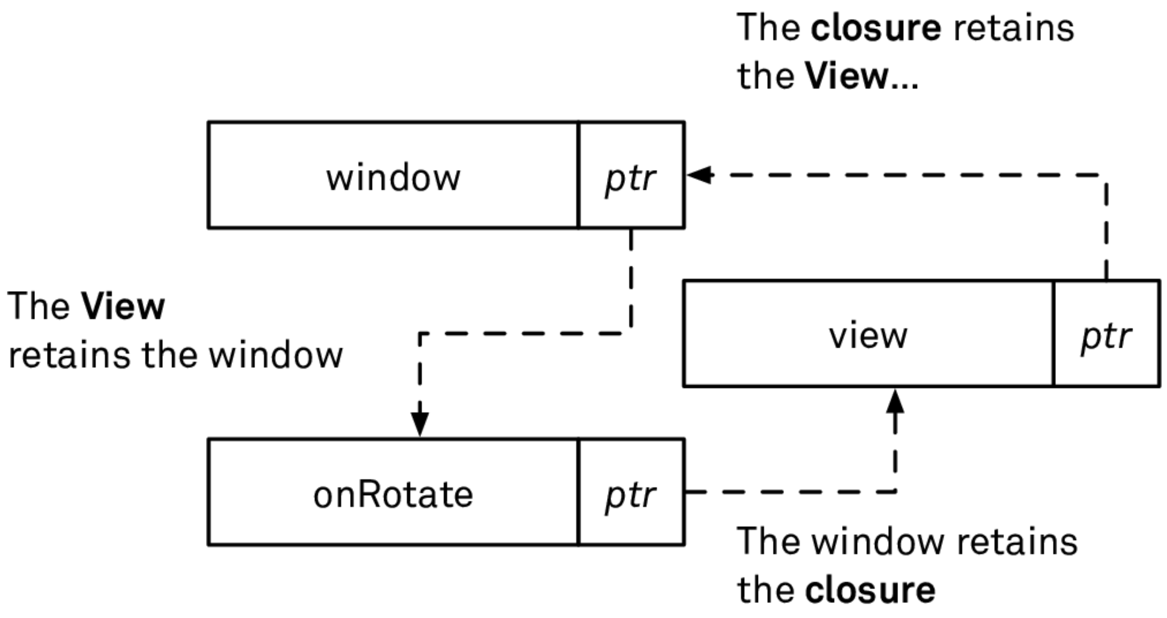
```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
```

The view has a strong reference to the window, but the window has a weak reference to the view, so there's no cycle. However, if we configure the onRotate callback and use the view in there, we've introduced a reference cycle:

```
window?.onRotate = {
    print("We now also need to update the view: \(view)")
}
```

The view references the window, the window references the callback, and the callback references the view: a cycle.

In a diagram, it looks like this:


The **closure** retains the **View**...
window
ptr
The **View** retains the window
view
ptr
onRotate
ptr
The window retains the **closure**

A retain cycle between the view, window, and closure

We need to find a way to break this cycle. There are three places where we could break the cycle (each corresponding to an arrow in the diagram):

- We could make the reference to the `Window` weak. Unfortunately, this would make the `Window` disappear, because there are no other references keeping it alive.

- We might want to change the `Window` to make the `onRotate` closure weak, but Swift doesn't allow marking closures as weak.

- We could make sure the closure doesn't reference the view by using a capture list. This is the only correct option in the above example.

In the case of the (constructed) example above, it's not too hard to figure out that we have a reference cycle. However, it's not always this easy. Sometimes the number of objects involved might be much larger, and the reference cycle might be harder to spot. And to make matters even worse, your code might be correct the first time you write it, but a refactoring might introduce a reference cycle without you noticing.

Capture Lists

To break the cycle above, we want to make sure the closure won't reference the view. We can do this by using a *capture list* and marking the captured variable (view) as either weak or unowned:

```
window?.onRotate = { [weak view] in
    print("We now also need to update the view: \(view)")
}
```

Capture lists can also be used to initialize new variables. For example, if we wanted to have a weak variable that refers to the window, we could initialize it in the capture list, or we could even define completely unrelated variables, like so:

```
window?.onRotate = { [weak view, weak myWindow=window, x=5*5] in
    print("We now also need to update the view: \(view)")
    print("Because the window \(myWindow) changed")
}
```

This is almost the same as defining the variable just above the closure, except that with capture lists, the scope of the variable is just the scope of the closure; it's not available outside of the closure.

Recap

We've looked at the differences between structs and classes in Swift. For entities (needing identity), classes are a better choice. For value types, structs are a better choice. When building structs that contain objects, we often need to take extra steps to ensure that they retain value semantics — for example, by implementing copy-on-write. We've looked at how to prevent reference cycles when dealing with classes. Often, a problem can be solved with either structs or classes, and what you choose depends on your needs. However, even problems that are classically solved using references can often benefit from values.

Encoding and Decoding

Serializing a program's internal data structures into some kind of data interchange format and vice versa is one of the most common programming tasks. Swift calls these operations *encoding* and *decoding*. One of the headline features of Swift 4 is a standardized design for encoding and decoding data that all custom types can opt into.

Overview

The `Codable` system (named after its base "protocol," which is really a type alias) is designed around three central goals:

- **Universality** — It should work with structs, enums, and classes.
- **Type safety** — Interchange formats such as JSON are often weakly typed, whereas your code should work with strongly typed data structures.
- **Reducing boilerplate** — Developers should have to write as little repetitive "adapter code" as possible to let custom types participate in the system. The compiler should generate this code for you automatically.

Types declare their ability to be (de-)serialized by conforming to the `Encodable` and/or `Decodable` protocols. Each of these protocols has just one requirement — `Encodable` defines an `encode(to:)` method in which a value encodes itself, and

Decodable specifies an initializer for creating an instance from serialized data:

```
/// A type that can encode itself to an external representation.
public protocol Encodable {
    /// Encodes this value into the given encoder.
    public func encode(to encoder: Encoder) throws
}

/// A type that can decode itself from an external representation.
public protocol Decodable {
    /// Creates a new instance by decoding from the given decoder.
    public init(from decoder: Decoder) throws
}
```

Because most types that adopt one will also adopt the other, the standard library provides the Codable typealias as a shorthand for both:

```
public typealias Codable = Decodable & Encodable
```

All basic standard library types — including Bool, the number types, and String — are codable out of the box, as are optionals, arrays, dictionaries, and sets containing codable elements. Lastly, many common data types used by Apple's frameworks have already adopted Codable, including Data, Date, URL, CGPoint, and CGRect.

Once you have a value of a codable type, you can create an *encoder* and tell it to serialize the value into the target format, such as JSON. In the reverse direction, a *decoder* takes serialized data and turns it back into an instance of the originating type. On the surface, the corresponding Encoder and Decoder protocols aren't much more complex than Encodable and Decodable. The central task of an encoder or decoder is to manage a hierarchy of *containers* that store the serialized data. While you rarely have to interact with the Encoder and Decoder protocols directly unless you're writing your own coders, understanding this structure and the three kinds of containers is necessary when you want to customize how your own types encode themselves. We'll see many examples of this below.

A Minimal Example

Let's begin with a minimal example of how you'd use the Codable system to encode an instance of a custom type to JSON.

Automatic Conformance

Making one of your own types codable can be as easy as conforming it to Codable. If all of the type's stored properties are themselves codable, the Swift compiler will automatically generate code that implements the Encodable and Decodable protocols. This Coordinate struct stores a GPS location:

```
struct Coordinate: Codable {
    var latitude: Double
    var longitude: Double
    // Nothing to implement here
}
```

Because both stored properties are already codable, adopting the Codable protocol is enough to satisfy the compiler. Similarly, we can now write a Placemark struct that takes advantage of Coordinate's Codable conformance:

```
struct Placemark: Codable {
    var name: String
    var coordinate: Coordinate
}
```

The code the compiler synthesizes isn't visible, but we'll dissect it piece by piece a little later in this chapter. For now, treat the generated code like you would a default implementation for a protocol in the standard library, such as Sequence.drop(while:) — you get the default behavior for free, but you have the option of providing your own implementation.

The only material difference between code generation and "normal" default implementations is that the latter are part of the standard library, whereas the logic for the Codable code synthesis lives in the compiler. Moving the code into the standard library would require more capable reflection APIs than Swift currently has, and even if those existed, runtime reflection would bring its own tradeoffs (e.g. reflection is typically slower).

Nonetheless, shifting as much of the language definition as possible out of the compiler and into libraries remains a stated goal for Swift. Some day we'll probably get a macro system that's powerful enough to move the entire Codable system into the standard library, but that's at least several years off. Until then, compiler code synthesis is a pragmatic solution to this problem. We're already seeing more applications of it for the upcoming automatic Equatable and Hashable conformance.

Encoding

Swift ships with two built-in encoders, JSONEncoder and PropertyListEncoder (these are defined in Foundation and not in the standard library). In addition, codable types are compatible with Cocoa's NSKeyedArchiver. We'll focus on JSONEncoder because JSON is the most common format.

Here's how we can encode an array of Placemark values to JSON:

```
let places = [
    Placemark(name: "Berlin", coordinate:
        Coordinate(latitude: 52, longitude: 13)),
    Placemark(name: "Cape Town", coordinate:
        Coordinate(latitude: -34, longitude: 18))
]
do {
    let encoder = JSONEncoder()
    let jsonData = try encoder.encode(places) // 129 bytes
    let jsonString = String(decoding: jsonData, as: UTF8.self)
/*
[{"name":"Berlin","coordinate":{"longitude":13,"latitude":52}},
{"name":"Cape Town","coordinate":{"longitude":18,"latitude":-34}}]
*/
} catch {
    print(error.localizedDescription)
}
```

The actual encoding step is exceedingly simple: create and (optionally) configure the encoder, and pass it the value to encode. The JSON encoder returns a collection of bytes in the form of a Data instance, which we then convert into a string for display.

In addition to a property for configuring the output formatting (pretty-printed and/or keys sorted lexicographically), JSONEncoder provides customization options for formatting dates (including ISO 8601 or Unix epoch timestamp) and Data values (e.g. Base64), as well as how exceptional floating-point values (infinity and *not a number*) should be treated. These options always apply to the entire hierarchy of values being encoded, i.e. you can't use them to specify that a Date in one type should follow a different encoding scheme than one in another type. If you need that kind of granularity, you have to write custom Codable implementations for the affected types.

It's worth noting that all of these configuration options are specific to JSONEncoder. Other encoders will have different options (or none at all). Even the encode(_:) method is encoder-specific and not defined in any of the protocols. Other encoders might decide to return a String or even a URL to the encoded file instead of a Data value.

In fact, JSONEncoder doesn't even conform to the Encoder protocol. Instead, it's a wrapper around a private class named _JSONEncoder that implements the protocol and does the actual encoding work. It was designed in this way because the top-level encoder should provide an entirely different API (namely, a single method to start the encoding process) than the Encoder object that's being passed to codable types during the encoding process. Separating these tasks cleanly means clients can only access the APIs that are appropriate in any given situation — for example, a codable type can't reconfigure the encoder in the middle of the encoding process because the public configuration API is only exposed by the top-level encoder.

Decoding

The decoding counterpart for JSONEncoder is JSONDecoder. Decoding follows the same pattern as encoding: create a decoder and pass it something to decode. JSONDecoder expects a Data instance containing UTF-8-encoded JSON text, but as we just saw with encoders, other decoders may have different interfaces:

```
do {
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark].self, from: jsonData)
// [Berlin (lat: 52.0, lon: 13.0), Cape Town (lat: -34.0, lon: 18.0)]
    type(of: decoded) // Array<Placemark>
    decoded == places // true
} catch {
    print(error.localizedDescription)
}
```

Notice that decoder.decode(_:from:) takes two arguments. In addition to the input data, we also have to specify the type we expect to get back (here it's [Placemark].self). This allows for full compile-time type safety. The tedious conversion from weakly typed JSON data to the concrete data types we use in our code happens behind the scenes.

Making the decoded type an explicit argument of the decoding method is a deliberate design choice. This wasn't strictly necessary, as the compiler would've been able to infer the correct type automatically in many situations, but the Swift team decided that the increase in clarity and avoidance of ambiguity was more important than maximum conciseness.

Even more so than during encoding, error handling is extremely important during decoding. There are so many things that can go wrong — from missing data (a required field is missing in the JSON input), to type mismatches (the server unexpectedly encodes numbers as strings), to fully corrupted data. Check out the documentation for the `DecodingError` type to see what other errors you can expect.

The Encoding Process

If *using* the `Codable` system is all you're interested in and the default behavior works for you, you can stop reading now. But to understand how to customize the way types get encoded, we need to dig a little deeper. How does the encoding process work? What code does the compiler actually synthesize when we conform a type to `Codable`?

When you initiate the encoding process, the encoder will call the `encode(to: Encoder)` method of the value that's being encoded, passing itself as the argument. It's then the value's responsibility to encode itself into the encoder in any format it sees fit.

In our example above, we pass an array of `Placemark` values to the JSON encoder:

```
let jsonData = try encoder.encode(places)
```

The encoder (or rather its private workhorse, `_JSONEncoder`) will now call `places.encode(to: self)`. How does the array know how to encode itself in a format the encoder understands?

Containers

Let's take a look at the `Encoder` protocol to see the interface an encoder presents to the value being encoded:

```
/// A type that can encode values into a native format for external representation.
public protocol Encoder {
    /// The path of coding keys taken to get to this point in encoding.
    var codingPath: [CodingKey] { get }
    /// Any contextual information set by the user for encoding.
    var userInfo: [CodingUserInfoKey : Any] { get }
    /// Returns an encoding container appropriate for holding
    /// multiple values keyed by the given key type.
    func container<Key: CodingKey>(keyedBy type: Key.Type)
        -> KeyedEncodingContainer<Key>
    /// Returns an encoding container appropriate for holding
    /// multiple unkeyed values.
    func unkeyedContainer() -> UnkeyedEncodingContainer
    /// Returns an encoding container appropriate for holding
    /// a single primitive value.
    func singleValueContainer() -> SingleValueEncodingContainer
}
```

Ignoring `codingPath` and `userInfo` for a moment, it's apparent that an `Encoder` is essentially a provider of *encoding containers*. A container is a sandboxed view into the encoder's storage. By creating a new container for each value that's being encoded, the encoder can make sure the values don't overwrite each other's data.

There are three types of containers:

- **Keyed containers** encode key-value pairs. Think of a keyed container as a special kind of dictionary. Keyed containers are by far the most prevalent.

  The keys in a keyed encoding container are strongly typed, providing type safety and autocompletion. The encoder will eventually convert the keys to strings (or integers) when it writes its target format (such as JSON), but that's hidden from client code. Changing the keys your type provides is the easiest way to customize how it encodes itself. We'll see an example of this below.

- **Unkeyed containers** encode multiple values sequentially, omitting keys. Think of an array of encoded values. Because there are no keys to identify a value, decoding containers must take care to decode values in the same order they were encoded.

- **Single value containers** encode a single value. You'd use this for types that are wholly defined by a single property. Examples include primitive types like `Int` or enums that are `RawRepresentable` as primitive values.

For each of the three container types, there's a protocol that defines the interface through which the container receives values to encode. Here's the definition of `SingleValueEncodingContainer`:

```swift
/// A container that can support the storage and direct encoding of a single
/// non-keyed value.
public protocol SingleValueEncodingContainer {
    /// The path of coding keys taken to get to this point in encoding.
    var codingPath: [CodingKey] { get }

    /// Encodes a null value.
    mutating func encodeNil() throws

    /// Base types
    mutating func encode(_ value: Bool) throws
    mutating func encode(_ value: Int) throws
    mutating func encode(_ value: Int8) throws
    mutating func encode(_ value: Int16) throws
    mutating func encode(_ value: Int32) throws
    mutating func encode(_ value: Int64) throws
    mutating func encode(_ value: UInt) throws
    mutating func encode(_ value: UInt8) throws
    mutating func encode(_ value: UInt16) throws
    mutating func encode(_ value: UInt32) throws
    mutating func encode(_ value: UInt64) throws
    mutating func encode(_ value: Float) throws
    mutating func encode(_ value: Double) throws
    mutating func encode(_ value: String) throws

    mutating func encode<T: Encodable>(_ value: T) throws
}
```

As you can see, the protocol mainly declares a bunch of encode(_:) overloads for various types: Bool, String, and the integer and floating-point types. There's also a variant for encoding a null value. Every encoder and decoder must support these primitive types, and all Encodable types must ultimately be reducible to one of these types. The Swift Evolution proposal that introduced the Codable system says:

> *These … overloads give strong, static type guarantees about what is encodable (preventing accidental attempts to encode an invalid type), and provide a list of primitive types that are common to all encoders and decoders that users can rely on.*

Any value that isn't one of the base types ends up in the generic encode<T: Encodable> overload. In it, the container will eventually call the argument's encode(to: Encoder) method, and the entire process will start over one level down until only primitive types are left. But the container is free to treat types with special requirements differently. For instance, it's at this point that _JSONEncoder checks if it's encoding a Data value that must observe the configured encoding strategy, such as Base64 (the default behavior for Data is to encode itself as an unkeyed container of UInt8 bytes).

UnkeyedEncodingContainer and KeyedEncodingContainerProtocol have the same structure as SingleValueEncodingContainer, but they expose a few additional capabilities, such as creating nested containers. If you want to write an encoder and decoder for another data format, implementing these containers is most of the work.

How a Value Encodes Itself

Going back to our example, the top-level type we're encoding is Array<Placemark>. An unkeyed container is a perfect match for an array (which is, after all, a sequential list of values), so the array asks the encoder for one. The array then iterates over its elements and tells the container to encode each one. Here's how this process looks in code:

```
extension Array: Encodable where Element: Encodable {
    public func encode(to encoder: Encoder) throws {
        var container = encoder.unkeyedContainer()
        for element in self {
            try container.encode(element)
        }
    }
}
```

(The actual code in the standard library is slightly longer and less elegant than this. The where Element: Encodable part doesn't compile in Swift 4 because the compiler doesn't support conditional protocol conformance yet. Runtime assertions and forced casts have to take its place.)

The array elements are Placemark instances. We've seen that for non-primitive types, the container will continue calling each value’s encode(to:) method.

The Synthesized Code

This brings us to the code the compiler synthesizes for the Placemark struct when we add Codable conformance. Let’s walk through it step by step.

Coding Keys

The first thing the compiler generates is a private nested enum named CodingKeys:

```
struct Placemark {
    // ...

    private enum CodingKeys: CodingKey {
        case name
        case coordinate
    }
}
```

The enum contains one case for every stored property of the struct. The enum values are the keys for a keyed encoding container. Compared to string keys, these strongly typed keys are much safer and more convenient to use because the compiler will find typos. However, encoders must eventually be able to convert keys to strings or integers for storage. Handling these translations is the task of the CodingKey protocol:

```
/// A type that can be used as a key for encoding and decoding.
public protocol CodingKey {
    /// The string to use in a named collection (e.g. a string-keyed dictionary).
    var stringValue: String { get }
    /// The value to use in an integer-indexed collection
    /// (e.g. an int-keyed dictionary).
    var intValue: Int? { get }
    init?(stringValue: String)
    init?(intValue: Int)
}
```

All keys must provide a string representation. Optionally, a key type can also provide a conversion to and from integers. Encoders can choose to use integer keys if that's more efficient, but they are also free to ignore them and stick with string keys (as JSONEncoder does). The default compiler-synthesized code only produces string keys.

The encode(to:) Method

This is the code the compiler generates for the Placemark struct's encode(to:) method:

```
struct Placemark: Codable {
    // ...
    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(name, forKey: .name)
        try container.encode(coordinate, forKey: .coordinate)
    }
}
```

The main difference to the array version is that Placemark encodes itself into a *keyed* container. A keyed container is the correct choice for all composite data types (structs and classes) with more than one property. Notice how the code passes CodingKeys.self to the encoder when it requests the keyed container. All subsequent encode commands into this container must specify a key of the same type. Since the key type is usually private to the type that's being encoded, this design makes it nearly impossible to accidentally use another type's coding keys when implementing this method manually.

The end result of the encoding process is a tree of nested containers which the JSON encoder can translate into its target format: keyed containers become JSON objects ({ … }), unkeyed containers become JSON arrays ([…]), and single value containers get converted to numbers, booleans, strings, or null, depending on their data type.

The init(from:) Initializer

When we call try decoder.decode([Placemark].self, from: jsonData), the decoder creates an instance of the type we passed in (here it's [Placemark]) using the initializer defined in Decodable. Like encoders, decoders manage a tree of *decoding containers*, which can be any of the three familiar kinds: keyed, unkeyed, or single value containers.

Each value being decoded then walks recursively down the container hierarchy and initializes its properties with the values it decodes from its container. If at any step an error is thrown (e.g. because of a type mismatch or missing value), the entire process fails with an error.

A typical decoding initializer implementation should look familiar to you. Here's the one the compiler generates for Placemark:

```
struct Placemark: Codable {
    // ...

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        name = try container.decode(String.self, forKey: .name)
        coordinate = try container.decode(Coordinate.self, forKey: .coordinate)
    }
}
```

Manual Conformance

If your type has special requirements, you can always implement the Encodable and Decodable requirements yourself. What's nice is that the automatic code synthesis isn't an all-or-nothing thing — you can pick and choose what to override and take the rest from the compiler.

Custom Coding Keys

The easiest way to control how a type encodes itself is to write a custom CodingKeys enum (which doesn't have to be an enum, by the way, although only enums get synthesized implementations for the CodingKey protocol). Providing custom coding keys is a very simple and declarative way of changing how a type is encoded. We can:

- **rename fields** in the encoded output by giving them an explicit string value, or
- **skip fields altogether** by omitting their keys from the enum.

To assign different names, we also have to give the enum an explicit raw value type of String. For example, this will map name to "label" in the JSON output while leaving the coordinate mapping unchanged:

```
struct Placemark2: Codable {
    var name: String
    var coordinate: Coordinate

    private enum CodingKeys: String, CodingKey {
        case name = "label"
        case coordinate
    }

    // Compiler-synthesized encode and decode methods
    // will use overridden CodingKeys
}
```

And this will skip the placemark's name and only encode the GPS coordinates because we didn't include the name key in the enum:

```
struct Placemark3: Codable {
    var name: String = "(Unknown)"
    var coordinate: Coordinate

    private enum CodingKeys: CodingKey {
        case coordinate
    }
}
```

Notice the default value we had to assign to the name property. Without it, code generation for Decodable will fail when the compiler detects that it can't assign a value to name in the initializer.

Skipping properties during encoding can be useful for transient values that can easily be recomputed or aren't important to store, such as caches or memoized expensive computations. The compiler is smart enough to filter out lazy properties on its own, but if you use normal stored properties for transient values, this is how you can do it yourself.

Custom encode(to:) and init(from:) Implementations

If you need more control, there's always the option to implement encode(to:) and/or init(from:) yourself. As an example, consider how the decoder deals with optional values. JSONEncoder and JSONDecoder can handle optionals out of the box. That is, if a property of the target type is optional, the decoder will correctly skip it if no corresponding value exists in the input data.

Here's an alternative definition of the Placemark type where the coordinate property is optional:

```
struct Placemark4: Codable {
    var name: String
    var coordinate: Coordinate?
}
```

Now our server can send us JSON data where the "coordinate" field is missing:

```
let validJSONInput = """
[
{ "name" : "Berlin" },
{ "name" : "Cape Town" }
]
"""
```

When we ask JSONDecoder to decode this input into an array of Placemark4 values, it'll automatically set the coordinate values to nil. So far so good.

However, JSONDecoder can be quite picky about the structure of the input data, and even small deviations from the expected format can trigger a decoding error. Now suppose the server is configured to send an empty JSON object to signify a missing optional value, so it sends this piece of JSON:

```
let invalidJSONInput = """
[
{
"name" : "Berlin",
"coordinate": {}
}
]
"""
```

When we try to decode this, the decoder, expecting the "latitude" and "longitude" fields inside the coordinate object, trips over the empty object and fails with a .keyNotFound error:

```
do {
    let inputData = invalidJSONInput.data(using: .utf8)!
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark4].self, from: inputData)
} catch {
    print(error.localizedDescription)
// The data couldn't be read because it is missing.
}
```

To make this work, we can override the Decodable initializer and explicitly catch the error we expect:

```
struct Placemark4: Codable {
    var name: String
    var coordinate: Coordinate?

    // encode(to:) is still synthesized by compiler

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.name = try container.decode(String.self, forKey: .name)
        do {
            self.coordinate = try container.decodeIfPresent(Coordinate.self,
                forKey: .coordinate)
        } catch DecodingError.keyNotFound {
            self.coordinate = nil
        }
    }
}
```

Now the decoder can successfully decode the faulty JSON:

```
do {
    let inputData = invalidJSONInput.data(using: .utf8)!
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark4].self, from: inputData)
    decoded // [Berlin (nil)]
} catch {
    print(error.localizedDescription)
}
```

Note that other errors, such as fully corrupted input data or any problem with the name field, will still throw.

This sort of customization is a nice option if only one or two types are affected, but it doesn't scale well. If a type has dozens of properties, you'll have to write manual code for each field, even if you only need to customize a single one. You might also want to read Dave Lyon's article on the topic. Dave came up with a generic protocol-based solution for exactly this issue. And if you have control over the input, it's always better to fix the problem at the source (make the server send valid JSON) rather than doctor with malformed data at a later stage.

Common Coding Tasks

In this section, we'd like to discuss some common tasks you might want to solve with the Codable system, along with potential problems you might run into.

Making Types You Don't Own Codable

Suppose we want to replace our Coordinate type with CLLocationCoordinate2D from the Core Location framework. CLLocationCoordinate2D has the exact same structure as Coordinate, so it makes sense not to reinvent the wheel.

The problem is that CLLocationCoordinate2D doesn't conform to Codable. As a result, the compiler will now (correctly) complain that it can't synthesize the Codable conformance for Placemark5 any longer because one of its properties isn't Codable itself:

```
import CoreLocation
```

```
struct Placemark5: Codable {
    var name: String
    var coordinate: CLLocationCoordinate2D
}
// error: cannot automatically synthesize 'Decodable'/'Encodable'
// because 'CLLocationCoordinate2D' does not conform
```

Can we make CLLocationCoordinate2D codable despite the fact that the type is defined in another module? Adding the missing conformance in an extension produces an error:

```
extension CLLocationCoordinate2D: Codable { }
// error: implementation of 'Decodable'/'Encodable' cannot be
// automatically synthesized in an extension yet
```

Swift 4.0 will only generate code for conformances that are specified on the type definition itself — we'd have to implement the protocols manually. But even if that limitation didn't exist (the Swift team plans to lift it, at least for extensions in the same file or module), retroactively adding Codable conformance to a type we don't own is probably not a good idea. What if Apple decides to provide the conformance itself in a future SDK version? It's likely that Apple's implementation won't be compatible with ours, which means values encoded with our version wouldn't decode with Apple's code, and vice versa. This is a problem because a decoder can't know which implementation it should use — it only sees that it's supposed to decode a value of type CLLocationCoordinate2D.

Itai Ferber, a developer at Apple who wrote large chunks of the Codable system, gives this advice:

> *I would actually take this a step further and recommend that any time you intend to extend someone else's type with Encodable or Decodable, you should almost certainly write a wrapper struct for it instead, unless you have reasonable guarantees that the type will never attempt to conform to these protocols on its own.*

We'll see an example of this in the next section. As for our current problem, let's go with a slightly different (but equally safe) solution: we'll provide our own Codable implementation for Placemark5, where we encode the latitude and longitude values directly. This effectively hides the existence of the CLLocationCoordinate2D type from the encoders and decoders; from their perspective, it looks as if the latitude and longitude properties were defined directly on Placemark5:

```
extension Placemark5 {
    private enum CodingKeys: String, CodingKey {
        case name
        case latitude = "lat"
        case longitude = "lon"
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(name, forKey: .name)
        // Encode latitude and longitude separately
        try container.encode(coordinate.latitude, forKey: .latitude)
        try container.encode(coordinate.longitude, forKey: .longitude)
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.name = try container.decode(String.self, forKey: .name)
        // Reconstruct CLLocationCoordinate2D from lat/lon
        self.coordinate = CLLocationCoordinate2D(
            latitude: try container.decode(Double.self, forKey: .latitude),
            longitude: try container.decode(Double.self, forKey: .longitude)
        )
    }
}
```

This example provides a good idea of the boilerplate code we'd have to write for every type if the compiler didn't generate it for us (and the synthesized implementation for the CodingKey protocol is still missing here).

Nested Containers

Alternatively, we could've used a *nested container* to encode the coordinates. KeyedDecodingContainer has a method named nestedContainer(keyedBy:forKey:) that creates a separate keyed container (with a separate coding key type) and stores it under the provided key. We'd add a second enum for the nested keys and encode the latitude and longitude values into the nested container (we're only showing the Encodable implementation here; Decodable follows the same pattern):

```
struct Placemark6: Encodable {
    var name: String
    var coordinate: CLLocationCoordinate2D

    private enum CodingKeys: CodingKey {
        case name
        case coordinate
    }

    // The coding keys for the nested container
    private enum CoordinateCodingKeys: CodingKey {
        case latitude
        case longitude
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(name, forKey: .name)
        var coordinateContainer = container.nestedContainer(
            keyedBy: CoordinateCodingKeys.self, forKey: .coordinate)
        try coordinateContainer.encode(coordinate.latitude, forKey: .latitude)
        try coordinateContainer.encode(coordinate.longitude, forKey: .longitude)
    }
}
```

With this approach, we would have effectively recreated the way the Coordinate type encodes itself inside our original Placemark struct, but without exposing the nested type to the Codable system at all. The resulting JSON is identical in both cases.

The Computed Property Workaround

As you can see, the amount of code we have to write for both alternatives is significant. For this particular example, we recommend a different approach, and that's to stick with our custom Coordinate struct for storage and Codable conformance, and expose the CLLocationCoordinate2D to clients as a computed property. Because the private _coordinate property is codable, we get the Codable conformance for free; all we have to do is rename its key in the CodingKeys enum. And the client-facing coordinate property has the type our clients require, but the Codable system will ignore it because it's a computed property:

```
struct Placemark7: Codable {
    var name: String
    private var _coordinate: Coordinate
    var coordinate: CLLocationCoordinate2D {
        get {
            return CLLocationCoordinate2D(latitude: _coordinate.latitude,
                longitude: _coordinate.longitude)
        }
        set {
            _coordinate = Coordinate(latitude: newValue.latitude,
                longitude: newValue.longitude)
        }
    }

    private enum CodingKeys: String, CodingKey {
        case name
        case _coordinate = "coordinate"
    }
}
```

This approach works well in this case because CLLocationCoordinate2D is such a simple type, and translating between it and our custom type is easy.

Making Classes Codable

We saw in the previous section that it's possible (if not advisable) to retroactively conform any value type to Codable. However, this isn't the case for non-final classes.

As a general rule, the Codable system works just fine with classes, but the potential existence of subclasses adds another level of complexity. What happens if we try to conform, say, UIColor to Decodable? (We're ignoring Encodable for the moment because it's not relevant for this discussion; we can add it later.) We got this example from a message by Jordan Rose on the swift-evolution mailing list.

A custom Decodable implementation for UIColor might look like this:

```
extension UIColor: Decodable {
    private enum CodingKeys: CodingKey {
        case red
        case green
        case blue
        case alpha
    }

    // Error: initializer requirement 'init(from:)' can only be satisfied
    // by a `required` initializer in the definition of non-final class 'UIColor'
    // etc.
    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        let red = try container.decode(CGFloat.self, forKey: .red)
        let green = try container.decode(CGFloat.self, forKey: .green)
        let blue = try container.decode(CGFloat.self, forKey: .blue)
        let alpha = try container.decode(CGFloat.self, forKey: .alpha)
        self.init(red: red, green: green, blue: blue, alpha: alpha)
    }
}
```

This code fails to compile, with several errors that ultimately boil down to one unsolvable conflict: only *required initializers* can satisfy protocol requirements, and required initializers may not be added in extensions; they must be declared directly in the class definition.

A required initializer (marked with the required keyword) indicates an initializer that every subclass must implement. The rule that initializers defined in protocols must be required ensures that they, like all protocol requirements, can be invoked dynamically on subclasses. The compiler must guarantee that code like this works:

```swift
func decodeDynamic(_ colorType: UIColor.Type,
    from decoder: Decoder) throws -> UIColor {
    return try colorType.init(from: decoder)
}
let color = decodeDynamic(SomeUIColorSubclass.self, from: someDecoder)
```

For this dynamic dispatch to work, the compiler must create an entry for the initializer in the class's dispatch table. This table of the class's non-final methods is created with a fixed size when the class definition is compiled; extensions can't add entries retroactively. This is why the required initializer is only allowed in the class definition.

Long story short: it's impossible to retroactively conform a non-final class to Codable in Swift 4. In the mailing list message we referenced above, Jordan Rose discusses a number of scenarios detailing how Swift could make this work in the future — from allowing a required initializer to be final (then it wouldn't need an entry in the dispatch table), to adding runtime checks that would trap if a subclass didn't provide the *designated initializer* that the required initializer calls.

But even then, we'd still have to deal with the fact that adding Codable conformance to types you don't own is problematic. As in the previous section, the recommended approach is to write a wrapper struct for UIColor and make that codable.

Let's start by writing a small extension that makes it easier to extract the red, green, blue, and alpha components from a UIColor value. The existing getRed(_:green:blue:alpha:) method uses pointers to pass the results back to the caller because Objective-C doesn't support tuples as return types. We can do better in Swift:

```
extension UIColor {
    var rgba: (red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)? {
        var red: CGFloat = 0.0
        var green: CGFloat = 0.0
        var blue: CGFloat = 0.0
        var alpha: CGFloat = 0.0
        if getRed(&red, green: &green, blue: &blue, alpha: &alpha) {
            return (red: red, green: green, blue: blue, alpha: alpha)
        } else {
            return nil
        }
    }
}
```

We'll use the rgba property in our encode(to:) implementation. Notice that the type of rgba is an optional tuple because not all UIColor instances can be represented as RGBA components. If someone tries to encode a color that isn't convertible to RGBA (e.g. a color based on a pattern image), we'll throw an encoding error.

Here's the complete implementation for our UIColor.CodableWrapper struct (we namespaced the struct inside UIColor to make the relationship between the two clear):

```
extension UIColor {
    struct CodableWrapper: Codable {
        var value: UIColor

        init(_ value: UIColor) {
            self.value = value
        }

        enum CodingKeys: CodingKey {
            case red
            case green
            case blue
            case alpha
        }

        func encode(to encoder: Encoder) throws {
            // Throw error if color isn't convertible to RGBA
            guard let (red, green, blue, alpha) = value.rgba else {
                let errorContext = EncodingError.Context(
                    codingPath: encoder.codingPath,
                    debugDescription:
                        "Unsupported color format: \(value)"
                )
                throw EncodingError.invalidValue(value, errorContext)
            }
            var container = encoder.container(keyedBy: CodingKeys.self)
            try container.encode(red, forKey: .red)
            try container.encode(green, forKey: .green)
            try container.encode(blue, forKey: .blue)
            try container.encode(alpha, forKey: .alpha)
        }

        init(from decoder: Decoder) throws {
            let container = try decoder.container(keyedBy: CodingKeys.self)
            let red = try container.decode(CGFloat.self, forKey: .red)
            let green = try container.decode(CGFloat.self, forKey: .green)
            let blue = try container.decode(CGFloat.self, forKey: .blue)
            let alpha = try container.decode(CGFloat.self, forKey: .alpha)
            self.value = UIColor(red: red, green: green, blue: blue, alpha: alpha)
        }
    }
}
```

We should note that this implementation is somewhat imperfect because it converts all colors to an RGB color space during encoding. When you later decode such a value, you'll always get back an RGB color, even if the color you originally encoded was in a grayscale color space, for example. Since there's no public UIColor API to identify the color space a color is in, a better implementation would have to drop down to the underlying CGColor to identify the color's color space model (e.g. RGB or grayscale) and then encode both the color space model and the components that make sense for the color space in question. On decoding, you'd then have to decode the color space model first before you'd know what other keys you could expect to find in the decoding container.

The big downside of the wrapper struct approach is that you'll have to manually convert between UIColor and the wrapper before encoding and after decoding. Suppose you want to encode an array of UIColor values:

```
let colors: [UIColor] = [
    .red,
    .white,
    .init(displayP3Red: 0.5, green: 0.4, blue: 1.0, alpha: 0.8),
    .init(hue: 0.6, saturation: 1.0, brightness: 0.8, alpha: 0.9),
]
```

You'll have to map the array to UIColor.CodableWrapper before you can pass it to an encoder:

```
let codableColors = colors.map(UIColor.CodableWrapper.init)
```

Moreover, any type that stores a UIColor will no longer participate in automatic code synthesis. Defining a type like the following produces an error because UIColor isn't Codable:

```
// error: cannot automatically synthesize 'Encodable'/'Decodable'
struct ColoredRect: Codable {
    var rect: CGRect
    var color: UIColor
}
```

How do we fix this with the least amount of code? As in the previous section, we'll add a private property of type UIColor.CodableWrapper that acts as storage for the color value, and we'll make color a computed property that forwards to _color in its accessors. We'll also need to add an initializer. Lastly, we'll provide our own coding keys enum to change the key used to encode the color value from the default "_color" to "color" (this step is optional):

```
struct ColoredRect: Codable {
    var rect: CGRect
    // Storage for color
    private var _color: UIColor.CodableWrapper
    var color: UIColor {
        get { return _color.value }
        set { _color.value = newValue }
    }

    init(rect: CGRect, color: UIColor) {
        self.rect = rect
        self._color = UIColor.CodableWrapper(color)
    }

    private enum CodingKeys: String, CodingKey {
        case rect
        case _color = "color"
    }
}
```

Encoding an array of ColoredRect values produces this JSON output:

```
let rects = [ColoredRect(rect: CGRect(x: 10, y: 20, width: 100, height: 200),
    color: .yellow)]
do {
    let encoder = JSONEncoder()
    let jsonData = try encoder.encode(rects)
    let jsonString = String(decoding: jsonData, as: UTF8.self)
// [{"color":{"red":1,"alpha":1,"blue":0,"green":1},"rect":[[10,20],[100,200]]}]
} catch {
    print(error.localizedDescription)
}
```

Making Enums Codable

The compiler can synthesize Codable conformance for RawRepresentable enums whose RawValue type is one of the "native" codable types, namely Bool, String, Float, Double, or one of the integer types. For other enums, such as those with associated values, you need to implement the requirements manually.

Let's conform the Either enum to Codable. Either is a very common type for modeling a value that can be *either* some value of a generic type A *or* some value of another type B. Here's the full implementation:

```
enum Either<A: Codable, B: Codable>: Codable {
    case left(A)
    case right(B)

    private enum CodingKeys: CodingKey {
        case left
        case right
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        switch self {
        case .left(let value):
            try container.encode(value, forKey: .left)
        case .right(let value):
            try container.encode(value, forKey: .right)
        }
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        if let leftValue = try container.decodeIfPresent(A.self, forKey: .left) {
            self = .left(leftValue)
        } else {
            let rightValue = try container.decode(B.self, forKey: .right)
            self = .right(rightValue)
        }
    }
}
```

In encode(to:), we check whether we have a *left* or *right* value and encode it under the corresponding key. Similarly, the init(from:) initializer uses the container's decodeIfPresent method to check if the container has a value for the *left* key. If not, it unconditionally decodes the *right* key, because one of the two keys must be present.

Ideally, we'd define the type as enum Either<A, B> — without any constraints on the generic type parameters — and then conditionally add the Codable conformance when both generic types are codable themselves, i.e. add extension Either: Codable where A: Codable, B: Codable. Unfortunately, the compiler doesn't support conditional conformances yet. We went for the easiest workaround, and that's to add the constraint directly to the type definition. This is good enough for an example; in production code, however, you probably can't afford to make Either incompatible with all non-codable payloads, so you'd have to resort to runtime checks. Array and other standard library types do this too.

Encoding and decoding a collection of values should be familiar by now. Let's use PropertyListEncoder and PropertyListDecoder for a change:

```
let values: [Either<String, Int>] = [
    .left("Forty-two"),
    .right(42)
]

do {
    let encoder = PropertyListEncoder()
    encoder.outputFormat = .xml
    let xmlData = try encoder.encode(values)
    let xmlString = String(decoding: xmlData, as: UTF8.self)
/*
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
<dict>
<key>left</key>
<string>Forty-two</string>
</dict>
<dict>
<key>right</key>
<integer>42</integer>
</dict>
</array>
</plist>
*/

    let decoder = PropertyListDecoder()
    let decoded = try decoder.decode([Either<String, Int>].self, from: xmlData)
/*
[Either<Swift.String, Swift.Int>.left("Forty-two"),
Either<Swift.String, Swift.Int>.right(42)]
*/
} catch {
    print(error.localizedDescription)
}
```

Decoding Polymorphic Collections

We've seen that decoders require us to pass in the *concrete type* of the value that's being decoded. This makes intuitive sense: the decoder needs a concrete type to figure out which initializer to call, and since the encoded data typically doesn't contain type information, the type must be provided by the caller. A consequence of this focus on strong typing is that there's no polymorphism in the decode step.

Suppose we want to encode an array of views, where the actual instances are UIView subclasses such as UILabel or UIImageView:

```
let views: [UIView] = [label, imageView, button]
```

(Let's assume for a moment that UIView and all its subclasses conform to Codable, which they currently don't.)

If we'd encode this array and then decode it again, it wouldn't come out in identical form — the concrete types of the array elements wouldn't survive. The decoder would only produce plain UIView objects because all it knows is that the type of the decoded data must be [UIView].self.

So how can we encode such a collection of polymorphic objects? The best option is to define an enum with one case per subclass we want to support. The payloads of the enum cases store the actual objects:

```
enum View {
    case view(UIView)
    case label(UILabel)
    case imageView(UIImageView)
    // ...
}
```

We then have to manually write a Codable implementation that follows a pattern similar to the Either enum in the previous section:

- During encoding, switch over all cases to figure out which type of object we need to encode. Then encode the object's type and the object itself under separate keys.

- During decoding, decode the type information first and select the concrete type to initialize based on that.

Finally, we should write two convenience functions for wrapping a UIView in a View value, and vice versa. That way, passing the source array to the encoder and getting it from the decoder only takes a single map.

Observe that this isn't a dynamic solution; we'll have to manually update the View enum every time we want to support another subclass. This is inconvenient, but it does make sense that we're forced to explicitly name every type our code can accept from a decoder. Anything else would be a potential security risk, because an attacker could use a manipulated archive to instantiate unknown objects in our program.

Recap

The ability to seamlessly convert between your program's native types and common data formats with minimal code — at least for the common case — is a very welcome addition to Swift. The Codable system becomes even more powerful if you can use Swift on both client and server: using the same types everywhere ensures all platforms will produce compatible encoding formats. Overriding the default behavior is always possible, if sometimes inconvenient when you need to handle non-Codable types you haven't defined yourself. The Swift team is already exploring more customization options, such as the ability to rewrite keys from camelCase to snake_case without having to touch every single type.

The system shines when you use it for the task it was designed for — working with uniform data in a known format in a fully type-safe manner. Swift does everything it can to hide the loosely typed dictionary-like data you'd get from a traditional JSON parser. In the rare cases where you'd rather work with a [String: Any] dictionary (maybe because you don't know the exact data format), don't try to intertwine the two worlds unnecessarily. The old-school method via JSONSerialization still exists.

We only discussed traditional archiving tasks in this chapter, but it's worth thinking outside the box to find other applications that can profit from a standardized way to reduce values to primitive data and vice versa. For example, you could write an encoder that computes a hash value from the encoded data and use that to automatically conform codable types to `Hashable`. This effectively uses the `Codable` system as a replacement for reflection. Or you could write a decoder that can produce random values for each of the primitive data types and use that to generate randomized test data for your unit tests.

Functions

To open this chapter, let's recap some main points regarding functions. If you're already very familiar with first-class functions, feel free to skip ahead to the next section. But if you're even slightly hazy, skim through what's below.

To understand functions and closures in Swift, you really need to understand three things, in roughly this order of importance:

0. Functions can be assigned to variables and passed in and out of other functions as arguments, just as an `Int` or a `String` can be.

1. Functions can *capture* variables that exist outside of their local scope.

2. There are two ways of creating functions — either with the `func` keyword, or with `{ }`. Swift calls the latter *closure expressions*.

Sometimes people new to the topic of closures come at it in reverse order and maybe miss one of these points, or conflate the terms *closure* and *closure expression* — and this can cause a lot of confusion. It's a three-legged stool, and if you miss one of the three points above, you'll fall over when you try to sit down.

1. Functions can be assigned to variables and passed in and out of other functions as arguments.

In Swift, as in many modern languages, functions are referred to as "first-class objects." You can assign functions to variables, and you can pass them in and out of other functions to be called later.

This is *the most important thing* to understand. "Getting" this for functional programming is akin to "getting" pointers in C. If you don't quite grasp this part, everything else will just be noise.

Let's start with a function that simply prints an integer:

```
func printInt(i: Int) {
    print("you passed\(i)")
}
```

To assign the function to a variable, funVar, we just use the function name as the value. Note the absence of parentheses after the function name:

```
let funVar = printInt
```

Now we can call the printInt function using the funVar variable. Note the use of parentheses after the variable name:

```
funVar(2) // you passed 2
```

It's also noteworthy that we *must not* include an argument label in the funVar call, whereas printInt calls *require* the argument label, as in printInt(i: 2). Swift only allows argument labels in function *declarations*; the labels aren't included in a function's *type*. This means that you currently can't assign argument labels to a variable of a function type, though this will likely change in a future Swift version.

We can also write a function that takes a function as an argument:

```
func useFunction(function: (Int) -> () ) {
    function(3)
}
```

```
useFunction(function: printInt) // you passed 3
useFunction(function: funVar) // you passed 3
```

Why is being able to treat functions like this such a big deal? Because it allows you to easily write "higher-order" functions, which take functions as arguments and apply them in useful ways, as we saw in the chapter on built-in collections.

Functions can also return other functions:

```
func returnFunc() -> (Int) -> String {
    func innerFunc(i: Int) -> String {
        return "you passed\(i)"
    }
    return innerFunc
}
let myFunc = returnFunc()
myFunc(3) // you passed 3
```

2. Functions can capture variables that exist outside of their local scope.

When a function references variables outside the function's scope, those variables are *captured* and stick around after they would otherwise fall out of scope and be destroyed.

To see this, let's revisit our returnFunc function but add a counter that increases each time we call it:

```
func counterFunc() -> (Int) -> String {
    var counter = 0
    func innerFunc(i: Int) -> String {
        counter += i // counter is captured
        return "running total:\(counter)"
    }
    return innerFunc
}
```

Normally counter, being a local variable of counterFunc, would go out of scope just after the return statement, and it'd be destroyed. Instead, because it's captured by innerFunc, the Swift runtime will keep it alive until the function that captured it gets destroyed. As we discussed in structs and classes, counter will exist on the heap rather than on the stack. We can call the inner function multiple times, and we see that the running total increases:

```
let f = counterFunc()
f(3) // running total: 3
f(4) // running total: 7
```

If we call counterFunc() again, a fresh counter variable will be created and captured:

```
let g = counterFunc()
g(2) // running total: 2
g(2) // running total: 4
```

This doesn't affect our first function, which still has its own captured version of counter:

```
f(2) // running total: 9
```

Think of these functions combined with their captured variables as similar to instances of classes with a single method (the function) and some member variables (the captured variables).

In programming terminology, a combination of a function and an environment of captured variables is called a *closure*. So f and g above are examples of closures, because they capture and use a non-local variable (counter) that was declared outside of them.

3. Functions can be declared using the { } syntax for closure expressions.

In Swift, you can define functions in two ways. One is with the func keyword. The other way is to use a *closure expression*. Consider this simple function to double a number:

```
func doubler(i: Int) -> Int {
  return i * 2
}
[1, 2, 3, 4].map(doubler) // [2, 4, 6, 8]
```

And here's the same function written using the closure expression syntax. Just like before, we can pass it to map:

```
let doublerAlt = { (i: Int) -> Int in return i*2 }
[1, 2, 3, 4].map(doublerAlt) // [2, 4, 6, 8]
```

Functions declared as closure expressions can be thought of as *function literals* in the same way that 1 and "hello" are integer and string literals. They're also anonymous — they aren't named, unlike with the func keyword. The only way they can be used is to assign them to a variable when they're created (as we do here with doubler), or to pass them to another function or method.

> *There's a third way anonymous functions can be used: you can call a function directly in line as part of the same expression that defines it. This can be useful for defining properties whose initialization requires more than one line. We'll see an example of this below in the section on lazy properties.*

The doubler declared using the closure expression, and the one declared earlier using the func keyword, are completely equivalent, apart from the differences in their handling of argument labels we mentioned above. They even exist in the same "namespace," unlike in some languages.

Why is the { } syntax useful then? Why not just use func every time? Well, it can be a lot more compact, especially when writing quick functions to pass into other functions, such as map. Here's our doubler map example written in a much shorter form:

```
[1, 2, 3].map { $0 * 2 } // [2, 4, 6]
```

This looks very different because we've leveraged several features of Swift to make code more concise. Here they are one by one:

0. If you're passing the closure in as an argument and that's all you need it for, there's no need to store it in a local variable first. Think of this like passing in a numeric expression, such as `5*i`, to a function that takes an `Int` as a parameter.

1. If the compiler can infer a type from the context, you don't need to specify it. In our example, the function passed to `map` takes an `Int` (inferred from the type of the array elements) and returns an `Int` (inferred from the type of the multiplication expression).

2. If the closure expression's body contains just a single expression, it'll automatically return the value of the expression, and you can leave off the `return`.

3. Swift automatically provides shorthand names for the arguments to the function — `$0` for the first, `$1` for the second, etc.

4. If the last argument to a function is a closure expression, you can move the expression outside the parentheses of the function call. This *trailing closure syntax* is nice if you have a multi-line closure expression, as it more closely resembles a regular function definition or other block statement such as `if expr { }`.

5. Finally, if a function has no arguments other than a closure expression, you can leave off the parentheses after the function name altogether.

Using each of these rules, we can boil down the expression below to the form shown above:

```
[1, 2, 3].map( { (i: Int) -> Int in return i * 2 } )
[1, 2, 3].map( { i in return i * 2 } )
[1, 2, 3].map( { i in i * 2 } )
[1, 2, 3].map( { $0 * 2 } )
[1, 2, 3].map() { $0 * 2 }
[1, 2, 3].map { $0 * 2 }
```

If you're new to Swift's syntax, and to functional programming in general, these compact function declarations might seem daunting at first. But as you get more comfortable with the syntax and the functional programming style, they'll start to feel more natural, and you'll be grateful for the ability to remove the clutter so you can see more clearly what the code is doing. Once you get used to reading code written like this, it'll be much clearer to you at a glance than the equivalent code written with a conventional for loop.

Sometimes, Swift needs a helping hand with inferring the types. And sometimes, you may get something wrong and the types aren't what you think they should be. If ever you get a mysterious error when trying to supply a closure expression, it's a good idea to write out the full form (first version above), complete with types. In many cases, that will help clear up where things are going wrong. Once you have the long form compiling, take the types out again one by one until the compiler complains. And if the error was yours, you'll have fixed your code in the process.

Swift will also insist you be more explicit sometimes. For example, you can't completely ignore input parameters. Suppose you wanted an array of random numbers. A quick way to do this is to map a range with a function that just generates random numbers. But you must supply an argument nonetheless. You can use _ in such a case to indicate to the compiler that you acknowledge there's an argument but that you don't care what it is:

```
(0..<3).map { _ in arc4random() } // [1461677845, 2240540000, 4124717241]
```

When you need to explicitly type the variables, you don't have to do it inside the closure expression. For example, try defining isEven without any types:

```
let isEven = { $0 % 2 == 0 }
```

Above, the type of isEven is inferred to be (Int) -> Bool in the same way that let i = 1 is inferred to be Int — because Int is the default type for integer literals.

This is because of a typealias, IntegerLiteralType, in the standard library:

```
protocol ExpressibleByIntegerLiteral {
    associatedtype IntegerLiteralType
    /// Create an instance initialized to value.
    init(integerLiteral value: Self.IntegerLiteralType)
}

/// The default type for an otherwise unconstrained integer literal.
typealias IntegerLiteralType = Int
```

If you were to define your own typealias, it would override the default one and change this behavior:

```
typealias IntegerLiteralType = UInt32
let i = 1 // i will be of type UInt32.
```

This is almost certainly a bad idea.

If, however, you needed a version of isEven for a different type, you could type the argument and return value inside the closure expression:

```
let isEvenAlt = { (i: Int8) -> Bool in i % 2 == 0 }
```

But you could also supply the context from *outside* the closure:

```
let isEvenAlt2: (Int8) -> Bool = { $0 % 2 == 0 }
let isEvenAlt3 = { $0 % 2 == 0 } as (Int8) -> Bool
```

Since closure expressions are most commonly used in some context of existing input or output types, adding an explicit type isn't often necessary, but it's useful to know.

Of course, it would've been much better to define a generic version of isEven that works on *any* integer as a computed property:

```
extension BinaryInteger {
    var isEven: Bool { return self % 2 == 0 }
}
```

Alternatively, we could have chosen to define an isEven variant for all Integer types as a free function:

```
func isEven<T: BinaryInteger>(_ i: T) -> Bool {
    return i % 2 == 0
}
```

If you want to assign that free function to a variable, this is also when you'd have to lock down which specific types it's operating on. A variable can't hold a generic function — only a specific one:

```
let int8isEven: (Int8) -> Bool = isEven
```

One final point on naming. It's important to keep in mind that functions declared with func can be closures, just like ones declared with { }. Remember, a closure is a function combined with any captured variables. While functions created with { } are called *closure expressions*, people often refer to this syntax as just *closures*. But don't get confused and think that functions declared with the closure expression syntax are different from other functions — they aren't. They're both functions, and they can both be closures.

Flexibility through Functions

In the built-in collections chapter, we talked about parameterizing behavior by passing functions as arguments. Let's look at another example of this: sorting.

Sorting a collection in Swift is simple:

```
let myArray = [3, 1, 2]
myArray.sorted() // [1, 2, 3]
```

There are really four sort methods: the non-mutating variant sorted(by:), and the mutating sort(by:), times two for the versions that default to sorting comparable things in ascending order and take no arguments. For the most common case, sorted() is all you need. And if you want to sort in a different order, just supply a function:

```
myArray.sorted(by: >) // [3, 2, 1]
```

You can also supply a function if your elements don't conform to Comparable but *do* have a < operator, like tuples:

```
var numberStrings = [(2, "two"), (1, "one"), (3, "three")]
numberStrings.sort(by: <)
numberStrings // [(1, "one"), (2, "two"), (3, "three")]
```

Or, you can supply a more complicated function if you want to sort by some arbitrary criteria:

```
let animals = ["elephant", "zebra", "dog"]
animals.sorted { lhs, rhs in
    let l = lhs.reversed()
    let r = rhs.reversed()
    return l.lexicographicallyPrecedes(r)
}
// ["zebra", "dog", "elephant"]
```

It's this last ability — the ability to use any comparison function to sort a collection — that makes the Swift sort so powerful.

Contrast this with the way sorting works in Objective-C. If you want to sort an array using Foundation, you're met with a long list of different options: there are sort methods that take a selector, block, or function pointer as a comparison predicate, or you can pass in an array of sort descriptors. All of these provide a lot of flexibility and power, but at the cost of complexity — there isn't an option to "just do a regular sort based on the default ordering." Some variants in Foundation, like the method that takes a block as a comparison predicate, are essentially the same as Swift's `sorted(by:)` method; others, like the version that takes an array of sort descriptors, make great use of Objective-C's dynamic nature to arrive at a very flexible and powerful (if weakly typed) API that can't be ported to Swift directly.

Support for selectors and dynamic dispatch is still there in Swift, but the Swift standard library favors a more function-based approach instead. In this section, we'll demonstrate how functions as arguments and treating functions as data can be used to replicate the same functionality in a fully type-safe manner. Let's look at a complex example inspired by the Sort Descriptor Programming Topics guide in Apple's documentation.

We'll start by defining a `Person` type. Because we want to show how Objective-C's powerful runtime system works, we'll have to make this object an `NSObject` subclass (in pure Swift, a struct might have been a better choice). We also annotate the class with `@objcMembers` to make all members visible to Objective-C:

```
@objcMembers
final class Person: NSObject {
    let first: String
    let last: String
    let yearOfBirth: Int
    init(first: String, last: String, yearOfBirth: Int) {
        self.first = first
        self.last = last
        self.yearOfBirth = yearOfBirth
    }
}
```

Let's also define an array of people with different names and birth years:

```
let people = [
    Person(first: "Emily", last: "Young", yearOfBirth: 2002),
    Person(first: "David", last: "Gray", yearOfBirth: 1991),
    Person(first: "Robert", last: "Barnes", yearOfBirth: 1985),
    Person(first: "Ava", last: "Barnes", yearOfBirth: 2000),
    Person(first: "Joanne", last: "Miller", yearOfBirth: 1994),
    Person(first: "Ava", last: "Barnes", yearOfBirth: 1998),
]
```

We want to sort this array first by last name, then by first name, and finally by birth year. The ordering should respect the user's locale settings. An NSSortDescriptor object describes how to order objects, and we can use them to express the individual sorting criteria (using localizedStandardCompare as the locale-conforming comparator method):

```
let lastDescriptor = NSSortDescriptor(key: #keyPath(Person.last),
    ascending: true,
    selector: #selector(NSString.localizedStandardCompare(_:)))
let firstDescriptor = NSSortDescriptor(key: #keyPath(Person.first),
    ascending: true,
    selector: #selector(NSString.localizedStandardCompare(_:)))
let yearDescriptor = NSSortDescriptor(key: #keyPath(Person.yearOfBirth),
    ascending: true)
```

To sort the array, we can use the sortedArray(using:) method on NSArray. This takes a list of sort descriptors. To determine the order of two elements, it starts by using the first sort descriptor and uses that result. However, if two elements are equal according to the first descriptor, it uses the second descriptor, and so on:

```
let descriptors = [lastDescriptor, firstDescriptor, yearDescriptor]
(people as NSArray).sortedArray(using: descriptors)
/*
[Ava Barnes (1998), Ava Barnes (2000), Robert Barnes (1985),
David Gray (1991), Joanne Miller (1994), Emily Young (2002)]
*/
```

A sort descriptor uses two runtime features of Objective-C: first, the key is an Objective-C key path, which is really just a string containing a chained list of property names. Don't confuse these with Swift's native (and strongly typed) key paths that were introduced in Swift 4. We'll have more to say about the latter below.

The second Objective-C runtime feature is key-value coding, which looks up the value of that key at runtime. The selector parameter takes a selector (which also is really just a string describing a method name). At runtime, the selector is used to look up a comparison function, and when comparing two objects, the values for the key are compared using that comparison function.

This is a pretty cool use of runtime programming, especially when you realize the array of sort descriptors can be built at runtime, based on, say, a user clicking a column heading.

How can we replicate this functionality using Swift's sort? It's simple to replicate *parts* of the sort — for example, if you want to sort an array using localizedStandardCompare:

```
var strings = ["Hello", "hallo", "Hallo", "hello"]
strings.sort { $0.localizedStandardCompare($1) == .orderedAscending }
strings // ["hallo", "Hallo", "hello", "Hello"]
```

If you want to sort using just a single property of an object, that's also simple:

```
people.sorted { $0.yearOfBirth < $1.yearOfBirth }
/*
[Robert Barnes (1985), David Gray (1991), Joanne Miller (1994),
Ava Barnes (1998), Ava Barnes (2000), Emily Young (2002)]
*/
```

This approach doesn't work so great when optional properties are combined with methods like localizedStandardCompare, though — it gets ugly fast. For example, consider sorting an array of filenames by file extension (using the fileExtension property from the optionals chapter):

```
var files = ["one", "file.h", "file.c", "test.h"]
files.sort { l, r in r.fileExtension.flatMap {
    l.fileExtension?.localizedStandardCompare($0)
} == .orderedAscending }
files // ["one", "file.c", "file.h", "test.h"]
```

This is quite ugly. Later on, we'll make it easier to use optionals when sorting. However, for now, we haven't even tried sorting by multiple properties. To sort by last name and then first name, we can use the standard library's lexicographicallyPrecedes method. This takes two sequences and performs a phonebook-style comparison by moving through each pair of elements until it finds one that isn't equal. So we can build two arrays of the elements and use lexicographicallyPrecedes to compare them. This method also takes a function to perform the comparison, so we'll put our use of localizedStandardCompare in the function:

```
people.sorted { p0, p1 in
    let left = [p0.last, p0.first]
    let right = [p1.last, p1.first]
    return left.lexicographicallyPrecedes(right) {
        $0.localizedStandardCompare($1) == .orderedAscending
    }
}
/*
[Ava Barnes (2000), Ava Barnes (1998), Robert Barnes (1985),
David Gray (1991), Joanne Miller (1994), Emily Young (2002)]
*/
```

At this point, we've almost replicated the functionality of the original sort in roughly the same number of lines. But there's still a lot of room for improvement: the building of arrays on every comparison is very inefficient, the comparison is hardcoded, and we can't really sort by yearOfBirth using this approach.

Functions as Data

Rather than writing an even more complicated function that we can use to sort, let's take a step back. So far, the sort descriptors were much clearer, but they use runtime programming. The functions we wrote don't use runtime programming, but they're not so easy to write (and read).

A sort descriptor is a way of describing the ordering of objects. Instead of storing that information as a class, we can define a function to describe the ordering of objects. The simplest possible definition takes two objects and returns true if they're ordered correctly. This is also exactly the type that the standard library's sort(by:) and sorted(by:) methods take as an argument. Let's define a generic type alias to describe sort descriptors:

```
/// A sorting predicate that returns true iff the first
/// value should be ordered before the second.
typealias SortDescriptor<Value> = (Value, Value) -> Bool
```

As an example, we could define a sort descriptor that compares two Person objects by year of birth, or a sort descriptor that sorts by last name:

```
let sortByYear: SortDescriptor<Person> = { $0.yearOfBirth < $1.yearOfBirth }
let sortByLastName: SortDescriptor<Person> = {
    $0.last.localizedStandardCompare($1.last) == .orderedAscending
}
```

Rather than writing the sort descriptors by hand, we can write a function that generates them. It's not nice that we have to write the same property twice: in sortByLastName, we could have easily made a mistake and accidentally compared $0.last with

$1.first. Also, it's tedious to write these sort descriptors; to sort by first name, it's probably easiest to copy and paste the sortByLastName definition and modify it.

Instead of copying and pasting, we can define a function with an interface that's a lot like NSSortDescriptor, but without the runtime programming. This function takes a key and a comparison function, and it returns a sort descriptor (the function, not the class NSSortDescriptor). Here, key isn't a string, but is itself a function; given an element of the array that's being sorted, it returns the value of the property this sort descriptor is dealing with. The areInIncreasingOrder function is then used to compare two keys. Finally, the return type is a function as well, even though this fact is slightly obscured by the type alias:

```
/// Builds a S or tDescrip → r function from a sorting predicate
/// and a key function which, given a value to compare, produces
/// the value that should be used by the sorting predicate.
func sortDescriptor<Value, Key>(
    key: @escaping (Value) -> Key,
    by areInIncreasingOrder: @escaping (Key, Key) -> Bool)
    -> SortDescriptor<Value>
{
    return { areInIncreasingOrder(key($0), key($1)) }
}
```

The key function describes how to drill down into a value and extract the information that is relevant for one particular sorting step. It has a lot in common with Swift's native key paths that were introduced in Swift 4. We'll discuss how to rewrite this with Swift key paths below.

This allows us to define sortByYear in a different way:

```
let sortByYearAlt: SortDescriptor<Person> =
    sortDescriptor(key: { $0.yearOfBirth }, by: <)
people.sorted(by: sortByYearAlt)
/*
[Robert Barnes (1985), David Gray (1991), Joanne Miller (1994),
Ava Barnes (1998), Ava Barnes (2000), Emily Young (2002)]
```

```
*/
```

We can even define an overloaded variant that works for all Comparable types:

```
func sortDescriptor<Value, Key>(key: @escaping (Value) -> Key)
    -> SortDescriptor<Value> where Key: Comparable
{
    return { key($0) < key($1) }
}
let sortByYearAlt2: SortDescriptor<Person> =
    sortDescriptor(key: { $0.yearOfBirth })
```

Both sortDescriptor variants above work with functions that return a boolean value, because that's the standard library's convention for comparison predicates. Foundation APIs like localizedStandardCompare, on the other hand, expect a three-way ComparisonResult value instead (ordered ascending, descending, or equal). It's possible the standard library will adopt this approach in the future. Adding support for this is easy as well:

```
func sortDescriptor<Value, Key>(
    key: @escaping (Value) -> Key,
    ascending: Bool = true,
    by comparator: @escaping (Key) -> (Key) -> ComparisonResult)
    -> SortDescriptor<Value>
{
    return { lhs, rhs in
        let order: ComparisonResult = ascending
            ? .orderedAscending
            : .orderedDescending
        return comparator(key(lhs))(key(rhs)) == order
    }
}
```

This allows us to write sortByFirstName in a much shorter and clearer way:

```
let sortByFirstName: SortDescriptor<Person> =
    sortDescriptor(key: { $0.first }, by: String.localizedStandardCompare)
people.sorted(by: sortByFirstName)
```

```
/*
[Ava Barnes (2000), Ava Barnes (1998), David Gray (1991),
Emily Young (2002), Joanne Miller (1994), Robert Barnes (1985)]
*/
```

This SortDescriptor is just as expressive as its NSSortDescriptor variant, but it's type safe, and it doesn't rely on runtime programming.

Currently, we can only use a single SortDescriptor function to sort arrays. If you recall, we used the NSArray.sortedArray(using:) method to sort an array with more than one comparison operator. We could easily add a similar method to Array, or even to the Sequence protocol. However, we'd have to add it twice: once for the mutating variant, and once for the non-mutating variant of sort.

We take a different approach so that we don't have to write more extensions. Instead, we write a function that combines multiple sort descriptors into a single sort descriptor. It works just like the sortedArray(using:) method: first it tries the first descriptor and uses that comparison result. However, if the result is equal, it uses the second descriptor, and so on, until we run out of descriptors:

```
func combine<Value>
    (sortDescriptors: [SortDescriptor<Value>]) -> SortDescriptor<Value> {
    return { lhs, rhs in
        for areInIncreasingOrder in sortDescriptors {
            if areInIncreasingOrder(lhs, rhs) { return true }
            if areInIncreasingOrder(rhs, lhs) { return false }
        }
        return false
    }
}
```

We can now finally replicate the initial example:

```
let combined: SortDescriptor<Person> = combine(
    sortDescriptors: [sortByLastName, sortByFirstName, sortByYear]
)
people.sorted(by: combined)
/*
[Ava Barnes (1998), Ava Barnes (2000), Robert Barnes (1985),
David Gray (1991), Joanne Miller (1994), Emily Young (2002)]
*/
```

We ended up with the same behavior and functionality as the Foundation version, but our solution is safer and a lot more idiomatic in Swift. Because the Swift version doesn't rely on runtime programming, the compiler can also optimize it much better. Additionally, we can use it with structs or non-Objective-C objects.

One drawback of the function-based approach is that functions are opaque. We can take an `NSSortDescriptor` and print it to the console, and we get some information about the sort descriptor: the key path, the selector name, and the sort order. Our function-based approach can't do this. If it's important to have that information, we could wrap the functions in a struct or class and store additional debug information.

This approach of using functions as data — storing them in arrays and building those arrays at runtime — opens up a new level of dynamic behavior, and it's one way in which a statically typed compile-time-oriented language like Swift can still replicate some of the dynamic behavior of languages like Objective-C or Ruby.

We also saw the usefulness of writing functions that combine other functions, which is one of the building blocks of functional programming. For example, our `combine(sortDescriptors:)` function took an array of sort descriptors and combined them into a single sort descriptor. This is a very powerful technique with many different applications.

Alternatively, we could even have written a custom operator to combine two sort functions:

```
infix operator <||> : LogicalDisjunctionPrecedence
func <||><A>(lhs: @escaping (A,A) -> Bool, rhs: @escaping (A,A) -> Bool)
    -> (A,A) -> Bool
{
    return { x, y in
        if lhs(x, y) { return true }
        if lhs(y, x) { return false }
        // Otherwise they're the same, so we check for the second condition
        if rhs(x, y) { return true }
        return false
    }
}
```

Most of the time, writing a custom operator is a bad idea. Custom operators are often harder to read than functions are, because the name isn't explicit. However, they can be very powerful when used sparingly. The operator above allows us to rewrite our combined sort example, like so:

```
let combinedAlt = sortByLastName <||> sortByFirstName <||> sortByYear
people.sorted(by: combinedAlt)
/*
[Ava Barnes (1998), Ava Barnes (2000), Robert Barnes (1985),
David Gray (1991), Joanne Miller (1994), Emily Young (2002)]
*/
```

This reads very clearly and perhaps also expresses the code's intent more succinctly than the alternative, but *only after* you (and every other reader of the code) have ingrained the meaning of the operator. We prefer the `combine(sortDescriptors:)` function over the custom operator. It's clearer at the call site and ultimately makes the code more readable. Unless you're writing highly domain-specific code, a custom operator is probably overkill.

The Foundation version still has one functional advantage over our version: it can deal with optionals without having to write any more code. For example, if we'd make the `last` property on `Person` an optional string, we wouldn't have to change anything in the sorting code that uses `NSSortDescriptor`.

The function-based version requires some extra code. You can probably guess what comes next: once again, we write a function that takes a function and returns a function. We can take a regular comparison function such as localizedStandardCompare, which works on two strings, and turn it into a function that takes two optional strings. If both values are nil, they're equal. If the left-hand side is nil, but the right-hand side isn't, they're ascending, and the other way around. Finally, if they're both non-nil, we can use the compare function to compare them:

```
func lift<A>(_ compare: @escaping (A) -> (A) -> ComparisonResult) -> (A?) -> (A?)
    -> ComparisonResult
{
    return { lhs in { rhs in
        switch (lhs, rhs) {
        case (nil, nil): return .orderedSame
        case (nil, _): return .orderedAscending
        case (_, nil): return .orderedDescending
        case let (l?, r?): return compare(l)(r)
        }
    }}
}
```

This allows us to "lift" a regular comparison function into the domain of optionals, and it can be used together with our sortDescriptor function. If you recall the files array from before, sorting it by fileExtension got really ugly because we had to deal with optionals. However, with our new lift function, it's very clean again:

```
let compare = lift(String.localizedStandardCompare)
let result = files.sorted(by: sortDescriptor(key: { $0.fileExtension },
    by: compare))
result // ["one", "file.c", "file.h", "test.h"]
```

> *We can write a similar version of lift for functions that return a Bool. As we saw in the [optionals](#) chapter, the standard library no longer provides comparison operators like > for optionals. They were removed because using them can lead to*

surprising results if you're not careful. A boolean variant of lift allows you to easily take an existing operator and make it work for optionals when you need the functionality.

This approach has also given us a clean separation between the sorting method and the comparison method. The algorithm that Swift's sort uses is a hybrid of multiple sorting algorithms — as of this writing, it's an introsort (which is itself a hybrid of a quicksort and a heapsort), but it switches to an insertion sort for small collections to avoid the upfront startup cost of the more complex sort algorithms.

Introsort isn't a "stable" sort. That is, it doesn't necessarily maintain relative ordering of values that are otherwise equal according to the comparison function. But if you implemented a stable sort, the separation of the sort method from the comparison would allow you to swap it in easily:

```
people.stablySorted(by: combine(
    sortDescriptors: [sortByLastName, sortByFirstName, sortByYear]
))
```

Local Functions and Variable Capture

If we wanted to implement such a stable sort, one choice might be a merge sort. The merge sort algorithm is made up of two parts: a division into sublists of one element, followed by a merge of those lists. Often, it's nice to define `merge` as a separate function. But this leads to a problem — `merge` requires some temporary scratch storage:

```
extension Array where Element: Comparable {
    private mutating func merge(lo: Int, mi: Int, hi: Int) {
        var tmp: [Element] = []
        var i = lo, j = mi
        while i != mi && j != hi {
            if self[j] < self[i] {
                tmp.append(self[j])
                j += 1
            } else {
                tmp.append(self[i])
                i += 1
            }
        }
        tmp.append(contentsOf: self[i..<mi])
        tmp.append(contentsOf: self[j..<hi])
        replaceSubrange(lo..<hi, with: tmp)
    }

    mutating func mergeSortInPlaceInefficient() {
        let n = count
        var size = 1
        while size < n {
            for lo in stride(from: 0, to: n-size, by: size*2) {
                merge(lo: lo, mi: (lo+size), hi: Swift.min(lo+size*2,n))
            }
            size *= 2
        }
    }
}
```

Note: because Array has a method named min() defined, we need to write Swift.min*. This tells the compiler to explicitly use the standard library's free function named min (instead of the method on Array).*

Of course, we could allocate this storage externally and pass it in as a parameter, but this is a little ugly. It's also complicated by the fact that arrays are value types — passing in an array we created outside wouldn't help. There's an optimization that replaces inout parameters with references, but the documentation tells us specifically not to rely on that.

As another solution, we can define merge as an inner function and have it capture the storage defined in the outer function's scope:

```
extension Array where Element: Comparable {
    mutating func mergeSortInPlace() {
        // Define the temporary storage for use by all merges
        var tmp: [Element] = []
        // and make sure it's big enough
        tmp.reserveCapacity(count)

        func merge(lo: Int, mi: Int, hi: Int) {
            // Wipe the storage clean while retaining its capacity
            tmp.removeAll(keepingCapacity: true)
            // The same code as before
            var i = lo, j = mi
            while i != mi && j != hi {
                if self[j] < self[i] {
                    tmp.append(self[j])
                    j += 1
                } else {
                    tmp.append(self[i])
                    i += 1
                }
            }
            tmp.append(contentsOf: self[i..<mi])
            tmp.append(contentsOf: self[j..<hi])
            replaceSubrange(lo..<hi, with: tmp)
        }

        let n = count
        var size = 1
        while size < n {
            for lo in stride(from: 0, to: n-size, by: size*2) {
                merge(lo: lo, mi: (lo+size), hi: Swift.min(lo+size*2,n))
            }
            size *= 2
        }
    }
}
```

Since closures (including inner functions) capture variables by reference, every call to merge within a single call to mergeSortInPlace will share this storage. But it’s still a local variable — separate concurrent calls to mergeSortInPlace will use separate instances. Using this technique can give a significant speed boost to the sort without needing major changes to the original version.

Functions as Delegates

Delegates. They're everywhere. Drummed into the heads of Objective-C (and Java) programmers is this message: use protocols (interfaces) for callbacks. You define a protocol, your owner implements that protocol, and it registers itself as your delegate so that it gets callbacks.

If a delegate protocol contains only a single method, you can mechanically replace the property storing the delegate object with one that stores the callback function directly. However, there are a number of tradeoffs to keep in mind.

Delegates, Foundation-Style

Let's start off by creating a protocol in the same way that Cocoa defines its countless delegate protocols. Most programmers who come from Objective-C have written code like this many times over:

```
protocol AlertViewDelegate: AnyObject {
    func buttonTapped(atIndex: Int)
}
```

It's defined as a class-only protocol because we want our AlertView class to hold a weak reference to the delegate. This way, we don't have to worry about reference cycles. An AlertView will never strongly retain its delegate, so even if the delegate (directly or indirectly) has a strong reference to the alert view, all is well. If the delegate is deinitialized, the delegate property will automatically become nil:

```
class AlertView {
    var buttons: [String]
    weak var delegate: AlertViewDelegate?

    init(buttons: [String] = ["OK", "Cancel"]) {
        self.buttons = buttons
    }

    func fire() {
        delegate?.buttonTapped(atIndex: 1)
    }
}
```

This pattern works really well when we're dealing with classes. For example, suppose we have a ViewController class that initializes the alert view and sets itself as the delegate. Because the delegate is marked as weak, we don't need to worry about reference cycles:

```
class ViewController: AlertViewDelegate {
    let alert: AlertView

    init() {
        alert = AlertView(buttons: ["OK", "Cancel"])
        alert.delegate = self
    }

    func buttonTapped(atIndex index: Int) {
        print("Button tapped: \(index)")
    }
}
```

It's common practice to always mark delegate properties as weak. This convention makes reasoning about the memory management very easy. Classes that implement the delegate protocol don't have to worry about creating a reference cycle.

Delegates That Work with Structs

Sometimes we might want to have a delegate protocol that's implemented by a struct. With the current definition of

AlertViewDelegate, this is impossible, because it's a class-only protocol.

We can loosen the definition of AlertViewDelegate by not making it a class-only protocol. Also, we'll mark the buttonTapped(atIndex:) method as mutating. This way, a struct can mutate itself when the method gets called:

```
protocol AlertViewDelegate {
    mutating func buttonTapped(atIndex: Int)
}
```

We also have to change our AlertView because the delegate property can no longer be weak:

```
class AlertView {
    var buttons: [String]
    var delegate: AlertViewDelegate?

    init(buttons: [String] = ["OK", "Cancel"]) {
        self.buttons = buttons
    }

    func fire() {
        delegate?.buttonTapped(atIndex: 1)
    }
}
```

If we assign an object to the delegate property, the object will be strongly referenced. Especially when working with delegates, the strong reference means there's a very high chance that we'll introduce a reference cycle at some point. However, we can use structs now. For example, we could create a struct that logs all button taps:

```
struct TapLogger: AlertViewDelegate {
    var taps: [Int] = []
    mutating func buttonTapped(atIndex index: Int) {
        taps.append(index)
    }
}
```

At first, it might seem like everything works well. We can create an alert view and a logger and connect the two. Alas, if we look at logger.taps after an event is fired, the array is still empty:

```
let alert = AlertView()
var logger = TapLogger()
alert.delegate = logger
alert.fire()
logger.taps // []
```

When we assigned to alert.delegate, Swift made a copy of the struct. So the taps aren't recorded in logger, but rather in alert.delegate. Even worse, when we assign the value, we lose the value's type. To get the information back out, we need to use a conditional type cast:

```
if let theLogger = alert.delegate as? TapLogger {
    print(theLogger.taps)
}
// [1]
```

Clearly this approach doesn't work well. When using classes, it's easy to create reference cycles, and when using structs, the original value doesn't get mutated. In short: delegate protocols don't make much sense when using structs.

Functions Instead of Delegates

If the delegate protocol only has a single method defined, we can simply replace the delegate property with a property that stores the callback function directly. In our case, this could be an optional buttonTapped property, which is nil by default:

```
class AlertView {
    var buttons: [String]
    var buttonTapped: ((_ buttonIndex: Int) -> ())?
    init(buttons: [String] = ["OK", "Cancel"]) {
        self.buttons = buttons
    }

    func fire() {
        buttonTapped?(1)
    }
}
```

The (_ buttonIndex: Int) -> () notation for the function type is a little weird because the internal name, buttonIndex, isn't relevant anywhere else in the code. We mentioned above that function types unfortunately can't have parameter labels; they *can* however have an explicit blank parameter label combined with an internal argument name. This is the officially sanctioned workaround to give parameters in function types labels for documentation purposes until Swift supports a better way.

Just like before, we can create a logger struct and then create an alert view instance and a logger variable:

```
struct TapLogger {
    var taps: [Int] = []

    mutating func logTap(index: Int) {
        taps.append(index)
    }
}

let alert = AlertView()
var logger = TapLogger()
```

However, we can't simply assign the logTap method to the buttonTapped property. The Swift compiler tells us that "partial application of a 'mutating' method is not allowed":

```
alert.buttonTapped = logger.logTap // Error
```

In the code above, it's not clear what should happen in the assignment. Does the logger get copied? Or should buttonTapped mutate the original variable (i.e. logger gets captured)?

To make this work, we have to wrap the right-hand side of the assignment in a closure. This has the benefit of making it very clear that we're now capturing the original logger variable (not the value) and that we're mutating it:

```
alert.buttonTapped = { logger.logTap(index: $0) }
```

As an additional benefit, the naming is now decoupled: the callback property is called buttonTapped, but the function that implements it is called logTap. Rather than a method, we could also specify an anonymous function:

```
alert.buttonTapped = { print("Button \($0) was tapped") }
```

When combining callbacks with classes, there are some caveats. Let's go back to our view controller example. In its initializer, instead of assigning itself as the alert view's delegate, the view controller can now assign its buttonTapped method to the alert view's callback handler:

```
class ViewController {
    let alert: AlertView

    init() {
        alert = AlertView(buttons: ["OK", "Cancel"])
        alert.buttonTapped = self.buttonTapped(atIndex:)
    }

    func buttonTapped(atIndex index: Int) {
        print("Button tapped: \(index)")
    }
}
```

The alert.buttonTapped = self.buttonTapped(atIndex:) line looks like an innocent assignment, but beware: we've just created a reference cycle! Every reference to an instance method of an object (like self.buttonTapped in the example) implicitly captures the object. To see why this has to be the case, consider the perspective of the alert view: when the alert view calls the callback function that's stored in its buttonTapped property, the function must somehow "know" which object's instance method it needs to call — it's not enough to just store a reference to ViewController.buttonTapped(atIndex:) without knowing the instance.

We could've shortened self.buttonTapped(atIndex:) to self.buttonTapped or just buttonTapped; all three refer to the same function. Parameter labels can be omitted as long as doing so doesn't create ambiguities.

It's very easy to accidentally create reference cycles this way. To avoid a strong reference, it's often necessary to wrap the method call in another closure that captures the object weakly:

```
alert.buttonTapped = { [weak self] index in
    self?.buttonTapped(atIndex: index)
}
```

This way, the alert view doesn't have a strong reference to the view controller. If we can guarantee that the alert view's lifetime is tied to the view controller, another option is to use [unowned self] instead of weak. With weak, should the alert view outlive the view controller, self will be nil inside the closure when the function gets called.

If you check out the type of the expression ViewController.buttonTapped, you'll notice that it's (ViewController) -> (Int) -> (). What's going on there? Under the hood, instance methods are modeled as functions that, given an instance, return another function that then operates on the instance. someVC.buttonTapped is really just another way of writing ViewController.buttonTapped(someVC) — both expressions return a function of type (Int) -> (), and this function is a closure that has strongly captured the someVC instance.

As we've seen, there are definite tradeoffs between protocols and callback functions. A protocol adds some verbosity, but a class-only protocol with a weak delegate removes the need to worry about introducing reference cycles.

Replacing the delegate with a function adds a lot of flexibility and allows you to use structs and anonymous functions. However, when dealing with classes, you need to be careful not to introduce a reference cycle.

Also, when you need multiple callback functions that are closely related (for example, providing the data for a table view), it can be helpful to keep them grouped together in a protocol rather than having individual callbacks. When using a protocol, a single type has to implement all the methods.

To unregister a delegate or a function callback, we can simply set it to `nil`. What about when our type stores an array of delegates or callbacks? With class-based delegates, we can simply remove an object from the delegate list. With callback functions, this isn't so simple; we'd need to add extra infrastructure for unregistering, because functions can't be compared.

inout Parameters and Mutating Methods

The "&" that we use at the front of an `inout` argument in Swift might give you the impression — especially if you have a C or C++ background — that `inout` parameters are essentially pass-by-reference. But they aren't. `inout` is pass-by-value-and-copy-back, *not* pass-by-reference. To quote the official Swift Programming Language book:

An inout parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value.

In the chapter on structs and classes, we wrote about inout parameters, and we looked at the similarities between mutating methods and methods that take an inout parameter.

In order to understand what kind of expressions can be passed as an inout parameter, we need to make the distinction between lvalues and rvalues. An *lvalue* describes a memory location. *lvalue* is short for "left value," because lvalues are expressions that can appear on the left side of an assignment. For example, array[0] is an lvalue, as it describes the memory location of the first element in the array. An *rvalue* describes a value. 2 + 2 is an rvalue, as it describes the value 4. You can't put 2 + 2 or 4 on the left side of an assignment statement.

For inout parameters, you can only pass lvalues, because it doesn't make sense to mutate an rvalue. When you're working with inout parameters in regular functions and methods, you need to be explicit about passing them in: every lvalue needs to be prefixed with an &. For example, when we call the increment function (which takes an inout Int), we can pass in a variable by prefixing it with an ampersand:

```
func increment(value: inout Int) {
    value += 1
}
var i = 0
increment(value: &i)
```

If we define a variable using let, we can't use it as an lvalue. This makes sense, because we're not allowed to mutate let variables; we can only use "mutable" lvalues:

```
let y: Int = 0
increment(value: &y) // Error
```

In addition to variables, a few more things are also lvalues. For example, we can also pass in an array subscript (if the array is defined using var):

```
var array = [0, 1, 2]
increment(value: &array[0])
array // [1, 1, 2]
```

In fact, this works with every subscript (including your own custom subscripts), as long as they both have a get and a set defined. Likewise, we can use properties as lvalues, but again, only if they have both get and set defined:

```
struct Point {
    var x: Int
    var y: Int
}
var point = Point(x: 0, y: 0)
increment(value: &point.x)
point // Point(x: 1, y: 0)
```

If a property is read-only (that is, only get is available), we can't use it as an inout parameter:

```
extension Point {
    var squaredDistance: Int {
        return x*x + y*y
    }
}
increment(value: &point.squaredDistance) // Error
```

Operators can also take an inout value, but for the sake of simplicity, they don't require the ampersand when called; we just specify the lvalue. For example, let's add back the postfix increment operator, which was removed in Swift 3:

```
postfix func ++(x: inout Int) {
    x += 1
}
point.x++
point // Point(x: 2, y: 0)
```

A mutating operator can even be combined with optional chaining. Here, we chain the increment operation to a dictionary subscript access:

```
var dictionary = ["one": 1]
dictionary["one"]?++
dictionary["one"] // Optional(2)
```

Note that the ++ operator won't get executed if the key lookup returns nil.

The compiler may optimize an inout variable to pass-by-reference, rather than copying in and out. However, it's explicitly stated in the documentation that we shouldn't rely on this behavior.

Nested Functions and inout

You can use an inout parameter inside nested functions, but Swift will make sure your usage is safe. For example, you can define a nested function (either using func or using a closure expression) and safely mutate an inout parameter:

```
func incrementTenTimes(value: inout Int) {
    func inc() {
        value += 1
    }
    for _ in 0..<10 {
        inc()
    }
}

var x = 0
incrementTenTimes(value: &x)
x // 10
```

However, you're not allowed to let that inout parameter escape (we'll talk more about escaping functions at the end of this chapter):

```
func escapeIncrement(value: inout Int) -> () -> () {
    func inc() {
        value += 1
    }
    // error: nested function cannot capture inout parameter
    // and escape
    return inc
}
```

This makes sense, given that the inout value is copied back just before the function returns. If we could somehow modify it later, what should happen? Should the value get copied back at some point? What if the source no longer exists? Having the compiler verify this is critical for safety.

When & Doesn't Mean inout

Speaking of unsafe functions, you should be aware of the other meaning of &: converting a function argument to an unsafe pointer.

If a function takes an UnsafeMutablePointer as a parameter, then you can pass a var into it using &, similar to an inout argument. But here you *really are* passing by reference — by pointer in fact.

Here's increment, written to take an unsafe mutable pointer instead of an inout:

```
func incref(pointer: UnsafeMutablePointer<Int>) -> () -> Int {
    // Store a copy of the pointer in a closure
    return {
        pointer.pointee += 1
        return pointer.pointee
    }
}
```

As we'll discuss in later chapters, Swift arrays implicitly decay to pointers to make C interoperability nice and painless. Now, suppose you pass in an array that goes out of scope before you call the resulting function:

```
let fun: () -> Int
do {
    var array = [0]
    fun = incref(pointer: &array)
}
fun()
```

This opens up a whole exciting world of undefined behavior. In testing, the above code printed different values on each run: sometimes 0, sometimes 1, sometimes 140362397107840 — and sometimes it produced a runtime crash.

The moral here is: know what you're passing in to. When appending an &, you could be invoking nice safe Swift inout semantics, or you could be casting your poor variable into the brutal world of unsafe pointers. When dealing with unsafe pointers, be very careful about the lifetime of variables. We'll go into more detail on this in the chapter on interoperability.

Properties

There are two special kinds of methods that differ from regular methods: computed properties and subscripts. A computed property looks like a regular property, but it doesn't use any memory to store its value. Instead, the value is computed on the fly every time the property is accessed. A computed property is really just a method with unusual defining and calling conventions.

Let’s look at the various ways to define properties. We’ll start with a struct that represents a GPS track. It stores all the recorded points in an array called record:

```
import CoreLocation

struct GPSTrack {
    var record: [(CLLocation, Date)] = []
}
```

If we want to make the record property available as read-only to the outside, but read-write internally, we can use the private(set) or fileprivate(set) modifiers:

```
struct GPSTrack {
    private(set) var record: [(CLLocation, Date)] = []
}
```

To access all the timestamps in a GPS track, we create a computed property:

```
extension GPSTrack {
    /// Returns all the timestamps for the GPS track.
    /// - Complexity: O(*n*), where *n* is the number of points recorded.
    var timestamps: [Date] {
        return record.map { $0.1 }
    }
}
```

Because we didn’t specify a setter, the timestamps property is read-only. The result isn’t cached; each time you access the property, it computes the result. The Swift API Design Guidelines recommend you document the complexity of every computed property that isn’t $O(1)$, because callers might assume that accessing a property takes constant time.

Change Observers

As we saw in the chapter on structs and classes, we can also implement the willSet and didSet handlers for properties and variables to get called every time a property is set (even if the value doesn't change). These get called immediately before and after the new value is stored, respectively. One useful case is when working with Interface Builder: we can implement didSet to know when an IBOutlet gets connected, and then perform additional configuration in the handler. For example, if we want to set a label's text color once it's available, we can do the following:

```
class SettingsController: UIViewController {
    @IBOutlet weak var label: UILabel? {
        didSet {
            label?.textColor = .black
        }
    }
}
```

The observers have to be defined at the declaration site of a property — you can't add one retroactively in an extension. They are therefore a tool for the *designer* of the type, and not the *user*. The willSet and didSet handlers are essentially shorthand for defining a pair of properties: one private stored property that provides the storage, and a public computed property whose setter performs additional work before and/or after storing the value in the stored property. This is fundamentally different from the key-value observing mechanism in Foundation, which is often used by *consumers* of an object to observe internal changes, whether or not the class's designer intended this.

You *can* override a property in a subclass to add an observer, however. Here's an example:

```
class Robot {
    enum State {
        case stopped, movingForward, turningRight, turningLeft
    }
    var state = State.stopped
}

class ObservableRobot: Robot {
    override var state: State {
        willSet {
            print("Transitioning from \(state) to \(newValue)")
        }
    }
}

var robot = ObservableRobot()
robot.state = .movingForward // Transitioning from stopped to movingForward
```

This is still consistent with the nature of change observers as an internal characteristic of a type. If it weren't allowed, a subclass could achieve the same effect by overriding a stored property with a computed setter that performs additional work.

The difference in usage is reflected in the implementation of these features. KVO uses the Objective-C runtime to dynamically add an observer to a class's setter, which would be impossible to implement in current Swift, especially for value types. Swift's property observers are a purely compile-time feature.

Lazy Stored Properties

Initializing a value lazily is such a common pattern that Swift has a special keyword, `lazy`, for defining lazy properties. Note that a lazy property must always be declared as `var` because its initial value might not be set until after initialization completes. Swift has a strict rule that `let` constants must have a value *before* an instance's initialization completes. The lazy modifier is a very specific form of memoization.

For example, if we have a view controller that displays a `GPSTrack`, we might want to have a preview image of the track. By making the property for that lazy, we can defer the expensive generation of the image until the property is accessed for the first time:

```
class GPSTrackViewController: UIViewController {
    var track: GPSTrack = GPSTrack()

    lazy var preview: UIImage = {
        for point in track.record {
            // Do some expensive computation
        }
        return UIImage(/* ... */)
    }()
}
```

Notice how we defined the lazy property: it's a closure expression that returns the value we want to store — in our case, an image. When the property is first accessed, the closure is executed (note the parentheses at the end), and its return value is stored in the property. This is a common pattern for lazy properties that require more than a one-liner to be initialized.

Because a lazy variable needs storage, we're required to define the lazy property in the definition of `GPSTrackViewController`. Unlike computed properties, stored properties and stored lazy properties can't be defined in an extension.

If the `track` property changes, the `preview` won't automatically get invalidated. Let's look at an even simpler example to see what's going on. We have a `Point` struct, and we store `distanceFromOrigin` as a lazy computed property:

```
struct Point {
    var x: Double
    var y: Double
    private(set) lazy var distanceFromOrigin: Double
        = (x*x + y*y).squareRoot()

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}
```

When we create a point, we can access the distanceFromOrigin property, and it'll compute the value and store it for reuse. However, if we then change the x value, this won't be reflected in distanceFromOrigin:

```
var point = Point(x: 3, y: 4)
point.distanceFromOrigin // 5.0
point.x += 10
point.distanceFromOrigin // 5.0
```

It's important to be aware of this. One way around it would be to recompute distanceFromOrigin in the didSet property observers of x and y, but then distanceFromOrigin isn't really lazy anymore: it'll get computed each time x or y changes. Of course, in this example, the solution is easy: we should have made distanceFromOrigin a regular (non-lazy) computed property from the beginning.

Accessing a lazy property is a mutating operation because the property's initial value is set on the first access. When a struct contains a lazy property, any owner of the struct that accesses the lazy property must therefore declare the struct as a variable too, because accessing the property means potentially mutating its container. So this isn't allowed:

```
let immutablePoint = Point(x: 3, y: 4)
immutablePoint.distanceFromOrigin
// Error: Cannot use mutating getter on immutable value
```

Forcing all users of the Point type who want to access the lazy property to use var is a huge inconvenience, which often makes lazy properties a bad fit for structs.

Also be aware that the lazy keyword doesn't perform any thread synchronization. If multiple threads access a lazy property at the same time before the value has been computed, it's possible the computation could be performed more than once, including any side effects the computation may have.

Early in Swift's open-source era, the Swift team proposed a general mechanism for annotating properties with "behaviors." If implemented, current primitive language features like property observers and lazy properties could be moved from the compiler into the standard library, making the compiler less complex and allowing all developers to add their own behaviors. The proposal was widely welcomed by the community but ultimately deferred due to time constraints. Something like property behaviors could still come in a future version of Swift.

Subscripts

We've already seen subscripts in the standard library. For example, we can perform a dictionary lookup like so: `dictionary[key]`. These subscripts are very much a hybrid of functions and computed properties, with their own special syntax. Like functions, they take arguments. Like computed properties, they can be either read-only (using `get`) or read-write (using `get set`). Just like normal functions, we can overload them by providing multiple variants with different types — something that isn't possible with properties. For example, arrays have two subscripts by default — one to access a single element, and one to get at a slice (to be precise, these are declared in the `Collection` protocol):

```
let fibs = [0, 1, 1, 2, 3, 5]
let first = fibs[0] // 0
fibs[1..<3] // [1, 1]
```

Custom Subscripts

We can add subscripting support to our own types, and we can also extend existing types with new subscript overloads. As an example, let's define a Collection subscript that takes a list of indices and returns an array of all elements at those indices:

```
extension Collection {
    subscript(indices indexList: Index...) -> [Element] {
        var result: [Element] = []
        for index in indexList {
            result.append(self[index])
        }
        return result
    }
}
```

Note how we used an explicit parameter label to disambiguate our subscript from those in the standard library. The three dots indicate that indexList is a *variadic parameter*. The caller can pass zero or more comma-separated values of the specified type (here, the collection's Index type). Inside the function, the parameters are made available as an array.

We can use the new subscript like this:

```
Array("abcdefghijklmnopqrstuvwxyz")[indices: 7, 4, 11, 11, 14]
// ["h", "e", "l", "l", "o"]
```

Advanced Subscripts

Subscripts aren't limited to a single parameter. We've already seen an example of a subscript that takes more than one parameter: the dictionary subscript that takes a key and a default value. Check out its implementation in the Swift source code if you're interested.

New in Swift 4, subscripts can now also be generic in their parameters or their return type. Consider a heterogeneous dictionary of type [String: Any]:

```
var japan: [String: Any] = [
    "name": "Japan",
    "capital": "Tokyo",
    "population": 126_740_000,
    "coordinates": [
        "latitude": 35.0,
        "longitude": 139.0
    ]
]
```

If you want to mutate a nested value in this dictionary, e.g. the coordinate's latitude, you'll find it isn't so easy:

```
// Error: Type 'Any' has no subscript members
japan["coordinate"]?["latitude"] = 36.0
```

OK, that's understandable. The expression japan["coordinate"] has the type Any?, so you'd probably try to cast it to a dictionary before applying the nested subscript:

```
// Error: Cannot assign to immutable expression
(japan["coordinates"] as? [String: Double])?["coordinate"] = 36.0
```

Alas, not only is this becoming ugly fast, but it doesn't work either. The problem is that you can't mutate a variable through a typecast — the expression japan["coordinates"] as? [String: Double] is no longer an lvalue. You'd have to store the nested dictionary in a local variable first, then mutate that variable, and then assign the local variable back to the top-level key.

We can do better by extending Dictionary with a generic subscript that takes the desired target type as a second parameter and attempts the cast inside the subscript implementation:

```swift
extension Dictionary {
    subscript<Result>(key: Key, as type: Result.Type) -> Result? {
        get {
            return self[key] as? Result
        }
        set {
            guard let value = newValue as? Value else {
                return
            }
            self[key] = value
        }
    }
}
```

Since we no longer have to downcast the value returned by the subscript, the mutation operation goes through to the top-level dictionary variable:

```swift
japan["coordinates", as: [String: Double].self]?["latitude"] = 36.0
japan["coordinates"] // Optional(["latitude": 36.0, "longitude": 139.0])
```

Generic subscripts close an important hole in the type system. That said, you'll notice the final syntax in this example is still quite ugly. Swift is generally not well suited for working on heterogeneous collections like this dictionary. In most cases, you'll be better off defining your own custom types for your data (e.g. here, a Country struct) and conforming those types to Codable for converting value to and from data transfer formats.

Key Paths

Swift 4 added the concept of *key paths* to the language. A key path is an uninvoked reference to a property, analogous to an unapplied method reference. Key paths close a fairly big hole in Swift's type system; previously, it wasn't possible to refer to a type's property, such as String.count, in the same way you could refer to a method, such as String.uppercased. Despite sharing the same name, Swift's key paths are quite different from the key paths used in Objective-C and Foundation. We'll have more to say about this later.

Key path expressions start with a backslash, e.g. \String.count. The backslash is necessary to distinguish a key path from a type property of the same name that may exist (suppose that String also had a static count property — then String.count would return the value of that property). Type inference works in key path expressions too: you can omit the type name if the compiler can infer it from the context, leaving \.count.

> *Given that key paths and function type references are so closely related, it's unfortunate that Swift has different syntaxes for the two. Even so, the Swift team has expressed interest in adopting the backslash syntax for function type references in a future release.*

As the name suggests, a key path describes a *path* through a value hierarchy, starting at the root value. For example, given the following Person and Address types, \Person.address.street is a key path that resolves a person's street address:

```
struct Address {
    var street: String
    var city: String
    var zipCode: Int
}

struct Person {
    let name: String
    var address: Address
}

let streetKeyPath = \Person.address.street // WritableKeyPath<Person, String>
let nameKeyPath = \Person.name // KeyPath<Person, String>
```

Key paths can be composed of any combination of stored and computed properties, along with optional chaining operators.

The compiler automatically generates a new [keyPath:] subscript for all types. You use this subscript to "invoke" the key path, i.e. to access the property described by it on a given instance. So "Hello"[keyPath: \.count] is equivalent to "Hello".count. Or, for our current example:

```
let simpsonResidence = Address(street: "1094 Evergreen Terrace",
    city: "Springfield", zipCode: 97475)
var lisa = Person(name: "Lisa Simpson", address: simpsonResidence)
lisa[keyPath: nameKeyPath] // Lisa Simpson
```

If you look at the types of our two key path variables above, you'll notice that the type of nameKeyPath is KeyPath<Person, String> (i.e. a strongly typed key path that can be applied to a Person and yields a String), whereas streetKeyPath's type is WritableKeyPath. Because all properties that form this latter key path are mutable, the key path itself allows mutation of the underlying value:

```
lisa[keyPath: streetKeyPath] = "742 Evergreen Terrace"
```

Doing the same with nameKeyPath would produce an error because the underlying property isn't mutable.

Key Paths Can Be Modeled with Functions

A key path that maps from a base type Root to a property of type Value is very similar to a function of type (Root) -> Value — or, for writable key paths, a *pair* of functions for both getting and setting a value. The major benefit key paths have over such functions (aside from the syntax) is that they are *values*. You can test key paths for equality and use them as dictionary keys (they conform to Hashable), and you can be sure that a key path is stateless, unlike functions, which might capture mutable state. None of these things are possible with normal functions.

Key paths are also composable by appending one key path to another. Notice that the types must match: if you start with a key path from A to B, the key path you append must have a root type of B, and the resulting key path will then map from A to the appended key path's value type, say C:

```
// KeyPath<Person, String> + KeyPath<String, Int> = KeyPath<Person, Int>
let nameCountKeyPath = nameKeyPath.appending(path: \.count)
// Swift.KeyPath<Person, Swift.Int>
```

Let's rewrite our sort descriptors from earlier in this chapter to use key paths instead of functions. We previously defined sortDescriptor as taking a function, (Key) -> Value:

```
typealias SortDescriptor<Value> = (Value, Value) -> Bool
func sortDescriptor<Value, Key>(key: @escaping (Value) -> Key)
    -> SortDescriptor<Value> where Key: Comparable {
    return { key($0) < key($1) }
}

// Usage
let streetSD: SortDescriptor<Person> = sortDescriptor { $0.address.street }
```

We can add a variant for constructing a sort descriptor from a key path. We use the subscript for key paths to access the value:

```
func sortDescriptor<Value, Key>(key: KeyPath<Value, Key>)
    -> SortDescriptor<Value> where Key: Comparable {
    return { $0[keyPath: key] < $1[keyPath: key] }
}

// Usage
let streetSDKeyPath: SortDescriptor<Person> =
    sortDescriptor(key: \.address.street)
```

While it's useful to have a sortDescriptor constructor that takes a key path, it doesn't give us the same flexibility as functions do. The key path relies on the Value being Comparable. Using just key paths, we can't easily sort by a different predicate (for example, performing a localized case-insensitive comparison).

Writable Key Paths

A writable key path is special; you can use it to read *or* write a value. Hence, it's equivalent to a pair of functions: one for getting ((Root) -> Value), and another for setting ((inout Root, Value) -> Void) the property. Writable key paths are a much bigger deal than the read-only key paths. First of all, they capture a lot of code in a neat syntax. Compare streetKeyPath with the equivalent getter and setter pair:

```
let streetKeyPath = \Person.address.street

let getStreet: (Person) -> String = { person in
    return person.address.street
}
let setStreet: (inout Person, String) -> () = { person, newValue in
    person.address.street = newValue
}

// Usage
lisa[keyPath: streetKeyPath] // 742 Evergreen Terrace
getStreet(lisa) // 742 Evergreen Terrace
```

Writable key paths are especially useful for *data binding*, where you want to *bind* two properties to each other: when property one changes, property two should automatically get updated, and vice versa. For example, you could bind a model.name property to textField.text. Users of the API need to specify how to read and write model.name and textField.text, and key paths capture just that.

We also need a way to observe changes to the properties. We use the key-value observing mechanism in Cocoa for this, which means that this example will only work with classes and only on Apple platforms. Foundation provides a new type-safe KVO API that hides the stringly typed world of Objective-C key paths. The NSObject method observe(_:options:changeHandler:) observes a key path (passed in as a strongly typed Swift key path) and calls the change handler whenever the property has changed. Don't forget to mark any property you want to observe as @objc dynamic, otherwise KVO won't work.

Our goal is to implement a two-way binding between two NSObjects, but let's start with a one-way binding: whenever the observed property of self changes, we change the other object as well. Key paths allow us to make this code generic over the specific properties involved: the caller specifies the two objects and the two key paths, and this method takes care of the rest:

```
extension NSObjectProtocol where Self: NSObject {
    func observe<A, Other>(_ keyPath: KeyPath<Self, A>,
        writeTo other: Other,
        _ otherKeyPath: ReferenceWritableKeyPath<Other, A>)
        -> NSKeyValueObservation
        where A: Equatable, Other: NSObjectProtocol
    {
        return observe(keyPath, options: .new) { _, change in
            guard let newValue = change.newValue,
                other[keyPath: otherKeyPath] != newValue else {
                    return // prevent endless feedback loop
            }
            other[keyPath: otherKeyPath] = newValue
        }
    }
}
```

There's a lot to unpack in this code snippet. First of all, we define this method on every subclass of NSObject, and by extending the NSObjectProtocol instead of NSObject, we're allowed to use Self. ReferenceWritableKeyPath is just like WritableKeyPath, but it also allows us to write reference variables (like other) that are declared using let. (We'll go into more details on that later.) To avoid unnecessary writes, we only write to other when the value has changed. The NSKeyValueObservation return value is a token the caller can use to control the lifetime of the observation:

observation stops either when this object gets deallocated or when the caller calls its invalidate method.

Given observe(_:writeTo:_:), two-way binding is straightforward: we call observe on both objects and return both observation tokens:

```
extension NSObjectProtocol where Self: NSObject {
    func bind<A, Other>(_ keyPath: ReferenceWritableKeyPath<Self,A>,
        to other: Other,
        _ otherKeyPath: ReferenceWritableKeyPath<Other,A>)
        -> (NSKeyValueObservation, NSKeyValueObservation)
        where A: Equatable, Other: NSObject
    {
        let one = observe(keyPath, writeTo: other, otherKeyPath)
        let two = other.observe(otherKeyPath, writeTo: self, keyPath)
        return (one,two)
    }
}
```

Now we can construct two different objects, Sample and MyObj, and bind the name and test properties to each other:

```
final class Sample: NSObject {
    @objc dynamic var name: String = ""
}

class MyObj: NSObject {
    @objc dynamic var test: String = ""
}

let sample = Sample()
let other = MyObj()
let observation = sample.bind(\Sample.name, to: other, \.test)
sample.name = "NEW"
other.test // NEW
other.test = "HI"
sample.name // HI
```

At the time of writing, key paths are a very new addition to Swift. We expect that many more interesting use cases will pop up in the future.

If you come from functional programming, writable key paths might remind you of lenses. They are closely related: from a `WritableKeypath<Root, Value>`*, you can create a* `Lens<Root, Value>`*. Lenses are useful in pure functional languages like Haskell or PureScript, but they aren't nearly as useful in Swift, because Swift has mutability built in.*

The Key Path Hierarchy

There are five different types for key paths, each adding more precision and functionality to the previous one:

- An `AnyKeyPath` is similar to a function `(Any) -> Any?`
- A `PartialKeyPath<Source>` is similar to a function `(Source) -> Any?`
- A `KeyPath<Source, Target>` is similar to a function `(Source) -> Target`
- A `WritableKeyPath<Source, Target>` is similar to a *pair* of functions: `(Source) -> Target` and `(inout Source, Target) -> ()`
- A `ReferenceWritableKeyPath<Source, Target>` is similar to a *pair* of functions: `(Source) -> Target` and `(Source, Target) -> ()`. The second function can update a `Source` value with `Target`, and works only when `Source` is a reference type. The distinction between `WritableKeyPath` and `ReferenceWritableKeyPath` is necessary because the setter for the former has to take its argument inout.

This hierarchy of key paths is currently implemented as a class hierarchy. Ideally, these would be protocols, but Swift 4's generics system lacks some features to make this feasible. The class hierarchy is intentionally kept closed to facilitate changing this in a future release without breaking existing code.

As we've seen before, key paths are different from functions: they conform to `Hashable`, and in the future they'll probably conform to `Codable` as well. This is why we say `AnyKeyPath` is *similar* to a function `(Any) -> Any`. While we can convert a key path into its corresponding function(s), we can't always go in the other direction.

Key Paths Compared to Objective-C

In Foundation and Objective-C, key paths are modeled as strings (we'll call these Foundation key paths to distinguish them from Swift's key paths). Because Foundation key paths are strings, they don't have any type information attached to them. In that sense, they're similar to `AnyKeyPath`. If a Foundation key path is misspelled, is not well formed, or if the types don't match, the program will probably crash. (The `#keyPath` directive in Swift helps a little with the misspelling; the compiler can check if a property with the specified name exists.) Swift's `KeyPath`, `WritableKeypath`, and `ReferenceWritableKeyPath` are correct by construction: they can't be misspelled and they don't allow for type errors.

Many Cocoa APIs use (Foundation) key paths when a function might have been better. This is partially a historical artifact: anonymous functions (or blocks, as Objective-C calls them) are a relatively recent addition, and key paths have been around much longer. Before blocks were added to Objective-C, it wasn't easy to represent something similar to the function `{ $0.address.street }`, except by using a key path: `"address.street"`.

Future Directions

Key paths are still under active discussion, and it's likely they'll become more capable in the future. One possible feature is serialization through the `Codable` protocol. This would allow us to persist key paths on disk, send them over the network, and so on. Once we have access to the structure of the key paths, this enables introspection. For example, we could use the structure of a key path to construct well-typed database queries. If types could automatically provide an array of key paths to their properties, this could serve as the basis for runtime reflection APIs.

Support for allowing subscript expressions in key paths is also planned, but the implementation didn't quite make the cut for Swift 4.0. In a future release, you'll probably be able to write things like `\[String].[n].count` to form a key path to an array's n-th element's character count, or `\[String: String].["name"]?.isEmpty` for a key path to a property of a value that's stored in a dictionary under the specified key. The Swift team has also expressed interest in supporting arbitrary function calls in key path expressions (provided those functions don't capture mutable state). For example, it's "not fair" that `\MyObject.firstFiveElements` and `\MyObject[3]` are valid KeyPaths, but `\MyObject.prefix(5)` isn't.

Automatic Closures

We're all familiar with how the logical AND operator `&&` evaluates its arguments. It first evaluates its left operand and immediately returns if that evaluation yields `false`. Only if the left operand evaluates to true is the right operand evaluated. After all, if the left operand evaluates to false, there's no way the entire expression can evaluate to true. This behavior is called *short-circuiting*. For example, if we want to check if a condition holds for the first element of an array, we could write the following code:

```
let evens = [2,4,6]
if !evens.isEmpty && evens[0] > 10 {
    // Perform some work
}
```

In the snippet above, we rely on short-circuiting: the array lookup happens only if the first condition holds. Without short-circuiting, this code would crash on an empty array.

> *A better way to write this particular example is to use an if let binding:*
>
> ```
> if let first = evens.first, first > 10 {
>     // Perform some work
> }
> ```
>
> *This is another form of short-circuiting: the second condition is only evaluated if the first one succeeds.*

In almost all languages, short-circuiting for the && and || operators is built into the language. However, it's often not possible to define your own operators or functions that have the same behavior. If a language supports first-class functions, we can fake short-circuiting by providing an anonymous function instead of a value. For example, let's say we wanted to define an and function in Swift with the same behavior as the && operator:

```
func and(_ l: Bool, _ r: () -> Bool) -> Bool {
    guard l else { return false }
    return r()
}
```

The function above first checks the value of l and returns false if l evaluates to false. Only if l is true does it return the value that comes out of the closure r. Using it is a little bit uglier than using the && operator, though, because the right operand now has to be a function:

```
if and(!evens.isEmpty, { evens[0] > 10 }) {
    // Perform some work
}
```

Swift has a nice feature to make this prettier. We can use the @autoclosure attribute to tell the compiler that it should wrap a particular argument in a closure expression. The definition of and is almost the same as above, except for the added @autoclosure annotation:

```
func and(_ l: Bool, _ r: @autoclosure () -> Bool) -> Bool {
    guard l else { return false }
    return r()
}
```

However, the usage of and is now much simpler, as we don't need to wrap the second parameter in a closure. We can just call it as if it took a regular Bool argument, and the compiler transparently wraps the argument in a closure expression:

```
if and(!evens.isEmpty, evens[0] > 10) {
    // Perform some work
}
```

This allows us to define our own functions and operators with short-circuiting behavior. For example, operators like ?? and !? (as defined in the chapter on optionals) are now straightforward to write. In the standard library, functions like assert and fatalError also use autoclosures in order to only evaluate the arguments when really needed. By deferring the evaluation of assertion conditions from the call sites to the body of the assert function, these potentially expensive operations can be stripped completely in optimized builds where they're not needed.

Autoclosures can also come in handy when writing logging functions. For example, here's how you could write your own `log` function, which only evaluates the log message if the condition is `true`:

```
func log(ifFalse condition: Bool,
    message: @autoclosure () -> (String),
    file: String = #file, function: String = #function, line: Int = #line)
{
    guard !condition else { return }
    print("Assertion failed:\(message()),\(file):\(function)(line\(line))")
}
```

This means you can perform expensive computations in the expression you pass as the message argument without incurring the evaluation cost if the value isn't used. The `log` function also uses the debugging identifiers `#file`, `#function`, and `#line`. They're especially useful when used as a default argument to a function, because they'll receive the values of the filename, function name, and line number at the call site.

Use autoclosures sparingly, though. Their behavior violates normal expectations — for example, if a side effect of an expression isn't executed because the expression is wrapped in an autoclosure. To quote Apple's Swift book:

> *Overusing autoclosures can make your code hard to understand. The context and function name should make it clear that evaluation is being deferred.*

The @escaping Annotation

As we saw in the previous chapter, we need to be careful about memory when dealing with functions. Recall the capture list example, where we needed to mark view as weak in order to prevent a reference cycle:

```
window?.onRotate = { [weak view] in
    print("We now also need to update the view: \(view)")
}
```

However, we never marked anything as weak when we used functions like map. This isn't necessary because the compiler knows that map will never create a reference cycle: it's executed synchronously and the closure never leaves map's scope. The difference between the closure we store in onRotate and the closure we pass to map is that the first closure *escapes*.

A closure that's stored somewhere to be called later (for example, after a function returns) is said to be *escaping*. The closure that gets passed to map only gets used directly within map. This means that the compiler doesn't need to change the reference count of the captured variables.

Closures are non-escaping by default. If you want to store a closure for later use, you need to mark the closure argument as @escaping. The compiler will verify this: unless you mark the closure argument as @escaping, it won't allow you to store the closure (or return it to the caller, for example). In the sort descriptors example, there were multiple function parameters that required the @escaping attribute:

```
func sortDescriptor<Value, Key>(
    key: @escaping (Value) -> Key,
    by areInIncreasingOrder: @escaping (Key, Key) -> Bool)
    -> SortDescriptor<Value>
{
    return { areInIncreasingOrder(key($0), key($1)) }
}
```

Before Swift 3, it was the other way around: escaping was the default, and you had the option to mark a closure as `@noescape`*. The current behavior is better because it's safe by default: a function argument now needs to be explicitly annotated to signal the potential for reference cycles. The* `@escaping` *annotation serves as a warning to the developer using the function. This is also the reason why the compiler enforces the use of* `self.` *for accessing members in escaping closures — the developer must explicitly opt into capturing a strong reference. Lastly, non-escaping closures can be optimized much better by the compiler, making the fast path the norm you have to explicitly deviate from if necessary.*

Note that the non-escaping-by-default rule only applies to function parameters, and then only for function types in *immediate parameter position*. This means stored properties that have a function type are always escaping (which makes sense). Surprisingly, the same is true for functions that *are* used as parameters, but are wrapped in some other type, such as a tuple or an optional. Since the closure is no longer an *immediate* parameter in this case, it automatically becomes escaping. As a consequence, you can't write a function that takes a function argument where the parameter is both optional and non-escaping. In many situations, you can avoid making the argument optional by providing a default value for the closure. If that's not possible, a workaround is to use overloading to write two variants of the function — one with an optional (escaping) function parameter, and one with a non-optional, non-escaping parameter:

```
func transform(_ input: Int, with f: ((Int) -> Int)?) -> Int {
    print("Using optional overload")
    guard let f = f else { return input }
    return f(input)
}

func transform(_ input: Int, with f: (Int) -> Int) -> Int {
    print("Using non-optional overload")
    return f(input)
}
```

This way, calling the function with a nil argument (or a variable of optional type) will use the optional variant, whereas passing a literal closure expression will invoke the non-escaping, non-optional overload.

```
transform(10, with: nil) // Using optional overload
transform(10) { $0 * $0 } // Using non-optional overload
```

withoutActuallyEscaping

You may run into a situation where you *know* that a closure doesn't escape but the compiler can't *prove* it, forcing you to add an @escaping annotation. In this example, we extend Array with a method that returns true if and only if all elements satisfy the given predicate. We already defined this method in the chapter on built-in collections.

Let's try a different implementation here: we first create a *lazy view* of the array (not to be confused with the lazy properties we discussed above) before applying a filter and checking whether any element got through the filter (i.e. whether at least one element didn't satisfy the predicate). The purpose of doing the operations lazily is that evaluation can stop as soon as the first mismatch is found. Our first attempt at implementing this fails to compile because the filter method on a lazy collection view takes an escaping closure:

```
extension Array {
    func all(matching predicate: (Element) -> Bool) -> Bool {
        // Error: parameter 'predicate' is implicitly non-escaping
        return self.lazy.filter({ !predicate($0) }).isEmpty
    }
}
```

We could fix this by annotating the parameter with @escaping, but in this case we *know* the closure won't escape because the lifetime of the lazy collection view is bound to the lifetime of the function. Swift provides an escape hatch for situations like this in the form of the withoutActuallyEscaping function. It allows you to pass a non-escaping closure to a function that expects an escaping one. This compiles and works correctly:

```
extension Array {
    func all(matching predicate: (Element) -> Bool) -> Bool {
        return withoutActuallyEscaping(predicate) { escapablePredicate in
            self.lazy.filter { !escapablePredicate($0) }.isEmpty
        }
    }
}
let areAllEven = [1,2,3,4].all { $0 % 2 == 0 } // false
let areAllOneDigit = [1,2,3,4].all { $0 < 10 } // true
```

Be aware that you're entering unsafe territory by using withoutActuallyEscaping. Allowing the copy of the closure to escape from the call to withoutActuallyEscaping results in undefined behavior.

Recap

Functions are first-class values in Swift. Treating functions as data can make our code more flexible. We've seen how we can replace runtime programming with simple functions. We've compared different ways to implement delegates. We've looked at mutating functions and inout parameters, as well as computed properties (which really are a special kind of function). Finally, we've discussed the @autoclosure and @escaping attributes. In the chapters on generics and protocols, we'll come up with more ways to use functions in Swift to gain additional flexibility.

Strings

All modern programming languages have support for Unicode strings, but that often only means that the native string type can store Unicode data — it's not a promise that simple operations like getting the length of a string will return "sensible" results. In fact, most languages, and in turn most string manipulation code written in those languages, exhibit a certain level of denial about Unicode's inherent complexity. This can lead to some unpleasant bugs.

Swift's string implementation goes to heroic efforts to be as Unicode-correct as possible. A `String` in Swift is a collection of `Character` values, where a `Character` is what a human reader of a text would perceive as a single character, regardless of how many Unicode code points it's composed of. As a result, all standard `Collection` operations like `count` or `prefix(5)` work on the level of user-perceived characters.

This is great for correctness, but it comes at a price, mostly in terms of unfamiliarity; if you're used to manipulating strings with integer indices in other languages, Swift's design will seem unwieldy at first, leaving you wondering. Why can't I write `str[999]` to access a string's one-thousandth character? Why doesn't `str[idx+1]` get the next character? Why can't I loop over a range of `Character` values such as `"a"..."z"`? It also has performance implications: `String` does *not* support random access, i.e. jumping to an arbitrary character is not an $O(1)$ operation. It can't be — when characters have variable width, the string doesn't know where the n^{th} character is stored without looking at all characters that come before it.

In this chapter, we'll discuss the string architecture in detail, as well as some techniques for getting the most out of Swift strings in terms of functionality and performance. But we'll start with an overview of the required Unicode terminology.

Unicode, Or: No More Fixed Width

Things used to be so simple. ASCII strings were a sequence of integers between 0 and 127. If you stored them in an 8-bit byte, you even had a bit to spare! Since every character was of a fixed size, ASCII strings could be random access.

But ASCII wasn't enough if you were writing in anything other than English or for a non-U.S. audience; other countries and languages needed other characters (even English-speaking Britain needed a £ sign). Most of them needed more characters than would fit into seven bits. ISO 8859 takes the extra bit and defines 16 different encodings above the ASCII range, such as Part 1 (ISO 8859-1, aka Latin-1), covering several Western European languages; and Part 5, covering languages that use the Cyrillic alphabet.

This is still limiting, though. If you want to use ISO 8859 to write in Turkish about Ancient Greek, you're out of luck, since you'd need to pick either Part 7 (Latin/Greek) or Part 9 (Turkish). And eight bits is still not enough to encode many languages. For example, Part 6 (Latin/Arabic) doesn't include the characters needed to write Arabic-script languages such as Urdu or Persian. Meanwhile, Vietnamese — which is based on the Latin alphabet but with a large number of diacritic combinations — only fits into eight bits by replacing a handful of ASCII characters from the lower half. And this isn't even an option for other East Asian languages.

When you run out of room with a fixed-width encoding, you have a choice: either increase the size, or switch to variable-width encoding. Initially, Unicode was defined as a 2-byte fixed-width format, now called UCS-2. This was before reality set in, and it was accepted that even two bytes would not be sufficient, while four would be horribly inefficient for most purposes.

So today, Unicode is a variable-width format, and it's variable in two different senses: in the combining of code units into Unicode scalars, and in the combining of scalars into characters.

Unicode data can be encoded with many different widths of *code unit,* most commonly 8 (UTF-8) or 16 (UTF-16) bits. UTF-8 has the added benefit of being backwardly compatible with 8-bit ASCII — something that's helped it overtake ASCII as the most popular encoding on the web. Swift represents UTF-16 and UTF-8 code units as `UInt16` and `UInt8` values, respectively (aliased as `Unicode.UTF16.CodeUnit` and `Unicode.UTF8.CodeUnit`).

A *code point* in Unicode is a single value in the Unicode code space with a possible value from `0` to `0x10FFFF` (in decimal: 1,114,111). Only about 137,000 of the 1.1 million available code points are currently in use, so there's a lot of room for more emoji. A given code point might take a single code unit if you're using UTF-32, or it might take between one and four if you're using UTF-8. The first 256 Unicode code points match the characters found in Latin-1.

Unicode *scalars* are almost, but not quite, the same as code points. They're all the code points *except* the 2,048 *surrogate code points* in the range `0xD800–0xDFFF`, i.e. the code points used for the leading and trailing codes that indicate pairs in UTF-16 encoding. Scalars are represented in Swift string literals as `"\u{xxxx}"`, where xxxx represents hex digits. So the euro sign can be written in Swift as either `"€"` or `"\u{20AC}"`. The corresponding

Swift type is `Unicode.Scalar`, which is a wrapper around a `UInt32` value.

To represent each Unicode scalar by a single code unit, you'd need a 21-bit encoding scheme (which usually gets rounded up to 32-bit, i.e. UTF-32), but even that wouldn't get you a fixed-width encoding: Unicode is still a variable-width format when it comes to "characters." What a user might consider "a single character" — as displayed on the screen — might require multiple scalars composed together. The Unicode term for such a user-perceived character is *(extended) grapheme cluster.*

The rules for how scalars form grapheme clusters determine how text is segmented. For example, if you hit the backspace key on your keyboard, you expect your text editor to delete exactly one grapheme cluster, even if that "character" is composed of multiple Unicode scalars, each of which may use a varying number of code units in the text's representation in memory. Grapheme clusters are represented in Swift by the `Character` type, which can encode an arbitrary number of scalars, as long as they form a single user-perceived character. We'll see some examples of this in the next section.

Grapheme Clusters and Canonical Equivalence

Combining Marks

A quick way to see how `String` handles Unicode data is to look at the two different ways to write é. Unicode defines U+00E9, *Latin small letter e with acute,* as a single value. But you can also write it as the plain letter e, followed by U+0301, *combining acute accent*. In both cases, what's displayed is é, and a user probably has a reasonable expectation that two strings displayed as "résumé" would not only be equal to each other but also have a "length" of six characters, no matter which technique was used to produce the é in either one. They would be what the Unicode specification describes as *canonically equivalent*.

And in Swift, this is exactly the behavior you get:

```
let single = "Pok\u{00E9}mon" // Pokémon
let double = "Poke\u{0301}mon" // Pokémon
```

They both display identically:

```
(single, double) // ("Pokémon", "Pokémon")
```

And both have the same character count:

```
single.count // 7
double.count // 7
```

Consequently, they also compare equal:

```
single == double // true
```

Only if you drop down to a view of the underlying representation can you see that they're different:

```
single.utf16.count // 7
double.utf16.count // 8
```

Contrast this with `NSString` in Foundation: the two strings aren't equal, and the `length` property — which many programmers probably use to count the number of characters to be displayed on the screen — gives different results:

```
let nssingle = single as NSString
nssingle.length // 7
let nsdouble = double as NSString
```

```
nsdouble.length // 8
nssingle == nsdouble // false
```

Here, == is defined as the version for comparing two NSObjects:

```
extension NSObject: Equatable {
    static func ==(lhs: NSObject, rhs: NSObject) -> Bool {
        return lhs.isEqual(rhs)
    }
}
```

In the case of NSString, this will do a literal comparison on the level of UTF-16 code units, rather than one accounting for equivalent but differently composed characters. Most string APIs in other languages work this way too. If you really want to perform a canonical comparison, you must use NSString.compare(_:). Didn't know that? Enjoy your future undiagnosable bugs and grumpy international user base.

Of course, there's one big benefit to just comparing code units: it's faster! This is an effect that can still be achieved with Swift strings, via the utf16 view:

```
single.utf16.elementsEqual(double.utf16) // false
```

Why does Unicode support multiple representations of the same character at all? The existence of precomposed characters is what enables the opening range of Unicode code points to be compatible with Latin-1, which already had characters like é and ñ. While they might be a pain to deal with, it makes conversion between the two encodings quick and simple.

And ditching precomposed forms wouldn't have helped anyway, because composition doesn't just stop at pairs; you can compose more than one diacritic together. For example, Yoruba has the character ọ́, which could be written three different ways: by composing ó with a dot, or by composing ọ with an acute, or by composing o with both an acute and a dot. And for that last one, the two diacritics can be in either order! So these are all equal:

```
let chars: [Character] = [
    "\u{1ECD}\u{300}", // ọ́
    "\u{F2}\u{323}", // ọ́
    "\u{6F}\u{323}\u{300}", // ọ́
    "\u{6F}\u{300}\u{323}" // ọ́
]
let allEqual = chars.dropFirst().all { $0 == chars.first } // true
```

(The all(matching:) method checks if the condition is true for all elements in a sequence and is defined in the chapter on built-in collections.)

In fact, some diacritics can be added ad infinitum. A famous internet meme illustrates this nicely:

```
let zalgo = "s̼̐͗͜o̠̦̤ͯͥ̒ͫ́ͅo̺̪͖̗̽ͩ̃͟ͅn̢͔͖͇͇͉̫̰ͪ͑"

zalgo.count // 4
zalgo.utf16.count // 36
```

In the above, zalgo.count (correctly) returns 4, while zalgo.utf16.count returns 36. And if your code doesn't work correctly with internet memes, then what good is it, really?

Unicode's grapheme breaking rules even affect you when all strings you deal with are pure ASCII: CR+LF, the character pair of carriage return and line feed that's commonly used as a line break on Windows, is a single grapheme:

```
// CR+LF is a single Character
let crlf = "\r\n"
crlf.count // 1
```

Emoji

Strings containing emoji can also be surprising in many other languages. Many emoji are assigned Unicode scalars that don't fit in a single UTF-16 code unit. Languages that represent strings as collections of UTF-16 code units, such as Java or C#, would say that the string "😂" is two "characters" long. Swift handles this case correctly:

```
let oneEmoji = "😂" // U+1F602
oneEmoji.count // 1
```

> *Notice that the important thing is how the string is exposed to the program, not how it's stored in memory. Swift also uses UTF-16 as its internal encoding for non-ASCII strings, but that's an implementation detail. The public API is based on grapheme clusters.*

Other emoji are composed of multiple scalars. An emoji flag is a combination of two regional indicator letters that correspond to an ISO country code. Swift treats it correctly as one `Character`:

```
let flags = "□□□□"
flags.count // 2
```

To inspect the Unicode scalars a string is composed of, use the `unicodeScalars` view. Here, we format the scalar values as hex numbers in the common format for code points:

```
flags.unicodeScalars.map {
    "U+\(String($0.value, radix: 16, uppercase: true))"
}
// ["U+1F1E7", "U+1F1F7", "U+1F1F3", "U+1F1FF"]
```

Skin tones combine a base character such as □ with one of five skin tone modifiers (e.g. □, or the type-4 skin tone modifier) to yield the final emoji (e.g. □□). Again, Swift handles this correctly:

```
let skinTone = "□□" // □ + □
skinTone.count // 1
```

This time, let's use a Foundation API to apply an ICU string transform that converts Unicode scalars to their official Unicode names:

```
extension StringTransform {
    static let toUnicodeName = StringTransform(rawValue: "Any-Name")
}

extension Unicode.Scalar {
    /// The scalar's Unicode name, e.g. "LATIN CAPITAL LETTER A".
    var unicodeName: String {
        // Force-unwrapping is safe because this transform always succeeds
        let name = String(self).applyingTransform(.toUnicodeName, reverse: false)!

        // The string transform returns the name wrapped in "\\N{...}". Remove those.
        let prefixPattern = "\\N{"
        let suffixPattern = "}"
        let prefixLength = name.hasPrefix(prefixPattern) ? prefixPattern.count : 0
        let suffixLength = name.hasSuffix(suffixPattern) ? suffixPattern.count : 0
        return String(name.dropFirst(prefixLength).dropLast(suffixLength))
    }
}

skinTone.unicodeScalars.map { $0.unicodeName }
// ["GIRL", "EMOJI MODIFIER FITZPATRICK TYPE-4"]
```

The essential part of this code snippet is the applyingTransform(.toUnicodeName, ...) call. The remaining lines clean up the name returned from the transform method by removing the wrapping braces. We code this defensively: we first check whether the string matches the expected pattern and compute the number of characters to strip from the start and end. If the format returned by the transform method changes in the future, it's better to return the string unchanged than to remove characters we didn't anticipate. Notice how we use the standard Collection methods dropFirst and dropLast to perform the stripping operation. This is a good example of how you can manipulate a string without doing manual index calculations. It's also efficient because dropFirst and dropLast return a Substring, which is a slice of the original string. No new memory allocations are needed until the final step when we create a new String from the substring. We'll have more to say about substrings later in this chapter.

Emoji depicting families and couples, such as □□□□ and □❤□, present another challenge to the Unicode standards body. Due to the countless possible combinations of genders and the number of people in a group, providing a separate code point for each variation is problematic. Combine this with a distinct skin tone for each person and it becomes impossible. Unicode solves this by specifying that these emoji are actually sequences of multiple emoji, combined with the invisible *zero-width joiner* (ZWJ) character (U+200D). So the family □□□□ is really *man* □ + ZWJ + *woman* □ + ZWJ + *girl* □ + ZWJ + *boy* □. The ZWJ serves as an indicator to the operating system that it should use a single glyph if available.

You can verify that this is really what's going on:

```
let family1 = "□□□□"
let family2 = "□\u{200D}□\u{200D}□\u{200D}□"
family1 == family2 // true
```

And once again, Swift is smart enough to treat such a sequence as a single `Character`:

```
family1.count // 1
family2.count // 1
```

New emoji for professions introduced in 2016 are ZWJ sequences too. For example, the *female firefighter* □□ is composed of *woman* □ + ZWJ + *fire engine* □, and the *male health worker* □⚕ is a sequence of *man* □ + ZWJ + *staff of aesculapius* ⚕.

Rendering these sequences into a single glyph is the task of the operating system. On Apple platforms in 2017, the OS includes glyphs for the subset of sequences the Unicode standard lists as "recommended for general interchange" (RGI), i.e. the ones "most likely to be widely supported across multiple platforms." When no glyph is available for a syntactically valid sequence, the text rendering system falls back to rendering each component as a separate glyph. Notice that this can cause a mismatch “in the other direction” between user-perceived characters and what Swift sees as a grapheme cluster; all examples up until now were concerned with programming languages *overcounting* characters, but here we see the reverse. As an example, family sequences containing skin tones are currently not part of the RGI subset. But even if the operating system renders such a sequence as multiple glyphs, Swift still counts it as a single Character because the Unicode text segmentation rules are not concerned with rendering:

```
// Family with skin tones is rendered as multiple glyphs
// on most platforms in 2017
let family3 = "□□\u{200D}□□\u{200D}□□\u{200D}□□"
// But Swift still counts it as a single Character
family3.count // 1
```

Microsoft can already render this and other variations as a single glyph, by the way, and the other OS vendors will almost certainly follow soon. But the point still stands: no matter how carefully a string API is designed, text is so complicated that it may never catch all edge cases.

> *In the past, Swift had trouble keeping up with Unicode changes. Swift 3 handled skin tones and ZWJ sequences incorrectly because its grapheme breaking algorithm was based on an old version of the Unicode standard. As of Swift 4, Swift uses the operating system’s ICU library. As a result, your programs will automatically adopt new Unicode rules as users update their OSes. The other side of the coin is of course that you can’t rely on users seeing the same behavior you see during development.*

In the examples we discussed in this section, we treated the length of a string as a proxy for all sorts of things that can go wrong when a language doesn't take the full complexity of Unicode into account. Just think of the gibberish a simple task such as reversing a string can produce in a programming language that doesn't process strings by grapheme clusters when the string contains composed character sequences. This isn't a new problem, but the emoji explosion has made it much more likely that bugs caused by sloppy text handling will come to the surface, even if your user base is predominantly English-speaking. And the magnitude of errors has increased as well: whereas a decade ago a botched accented character would cause an off-by-one error, messing up a modern emoji can easily cause results to be off by 10 or more "characters." For example, a four-person family emoji is 11 (UTF-16) or 25 (UTF-8) code units long:

```
family1.count // 1
family1.utf16.count // 11
family1.utf8.count // 25
```

It's not that other languages don't have Unicode-correct APIs at all — most do. For instance, NSString has the enumerateSubstrings method that can be used to walk through a string by grapheme clusters. But defaults matter; Swift's priority is to do the correct thing by default. And if you ever need to drop down to a lower level of abstraction, String provides views that let you operate directly on Unicode scalars or code units. We'll say more about those below.

Strings and Collections

As we've seen, String is a collection of Character values. In Swift's first three years of existence, String went back and forth between conforming and not conforming to the Collection protocol. The argument for *not* adding the conformance was that programmers would expect all generic collection-processing algorithms to be completely safe and Unicode-correct, which wouldn't necessarily be true for all edge cases.

As a simple example, you might assume that if you concatenate two collections, the resulting collection's length would be the sum of the lengths of the two source collections. But this doesn't hold for strings if a suffix of the first string forms a grapheme cluster with a prefix of the second string:

```
let flagLetterJ = "🇯"
let flagLetterP = "🇵"
let flag = flagLetterJ + flagLetterP // 🇯🇵
flag.count // 1
flag.count == flagLetterJ.count + flagLetterP.count // false
```

To this end, `String` itself was not made a `Collection` in Swift 2 and 3; a collection-of-characters view was moved to a property, `characters`, which put it on a footing similar to the other collection views: `unicodeScalars`, `utf8`, and `utf16`. Picking a specific view prompted you to acknowledge you were moving into a "collection-processing" mode and that you should consider the consequences of the algorithm you were about to run.

In practice, the loss in usability and learnability caused by this change turned out to vastly outweigh the gain in correctness for a few edge cases that are rarely relevant in real code (unless you're writing a text editor). So `String` was made a `Collection` again in Swift 4. The `characters` view still exists, but only for backward compatibility.

Bidirectional, Not Random Access

However, for reasons that should be clear from the examples in the previous section, `String` is *not* a random-access collection. How could it be, when knowing where the n^{th} character of a particular string is involves evaluating just how many Unicode scalars precede that character? For this reason, String conforms only to BidirectionalCollection. You can start at either end of the string, moving forward or backward, and the code will look at the composition of the adjacent characters and skip over the correct number of bytes. However, you need to iterate up and down one character at a time.

Keep the performance implications of this in mind when writing string-processing code. Algorithms that depend on random access to maintain their performance guarantees aren't a good match for Unicode strings. Consider this String extension for generating a list of a string's prefixes, which works by generating an integer range from zero to the string's length and then mapping over the range to create the prefix for each length:

```
extension String {
    var allPrefixes1: [Substring] {
        return (0...self.count).map(self.prefix)
    }
}

let hello = "Hello"
hello.allPrefixes1 // ["", "H", "He", "Hel", "Hell", "Hello"]
```

As simple as this code looks, it's very inefficient. It first walks over the string once to calculate the length, which is fine. But then each of the n + 1 calls to prefix is another $O(n)$ operation because prefix always starts at the beginning and has to work its way through the string to count the desired number of characters. Running a linear process inside another linear loop means this algorithm is accidentally $O(n^2)$ — as the length of the string increases, the time this algorithm takes increases quadratically.

If possible, an efficient string algorithm should walk over a string only once and then operate on string indices to denote the substrings it's interested in. Here's another version of the same algorithm:

```
extension String {
    var allPrefixes2: [Substring] {
        return [""] + self.indices.map { index in self[...index] }
    }
}

hello.allPrefixes2 // ["", "H", "He", "Hel", "Hell", "Hello"]
```

This code also has to iterate over the string once to generate the indices collection. But once that's done, the subscripting operation inside map is $O(1)$. This makes the whole algorithm $O(n)$.

> *The PrefixIterator we implemented in the collection protocols chapter solves the same problem in a generic way for all sequences.*

Range-Replaceable, Not Mutable

String also conforms to RangeReplaceableCollection. Here's an example of how you'd replace part of a string by first identifying the appropriate range in terms of string indices and then calling replaceSubrange. The replacement string can have a different length or could even be empty (which would be equivalent to calling removeSubrange):

```
var greeting = "Hello, world!"
if let comma = greeting.index(of: ",") {
    greeting[..<comma] // Hello
    greeting.replaceSubrange(comma..., with: " again.")
}
greeting // Hello again.
```

As always, keep in mind that results may be surprising if parts of the replacement string form new grapheme clusters with adjacent characters in the original string.

One collection-like feature strings do *not* provide is that of MutableCollection. This protocol adds one feature to a collection — that of the single-element subscript set — in addition to get. This isn't to say strings aren't mutable — as we've just seen, they have several mutation methods. But what you can't do is replace a single character using the subscript operator. The reason comes back to variable-length characters. Most people can probably intuit that a single-element subscript update would happen in constant time, as it does for Array. But since a character in a string may be of variable width, updating a single character could take linear time in proportion to the length of the string: changing the width of a single element would require shuffling all the later elements up or down in memory. Moreover, indices that come after the replaced index would become invalid through the shuffling, which is equally unintuitive. For these reasons, you have to use replaceSubrange, even if the range you pass in is only a single element.

String Indices

Most programming languages use integers for subscripting strings, e.g. str[5] would return the sixth "character" of str (for whatever that language's idea of a "character" is). Swift doesn't allow this. Why? The answer should sound familiar to you by now: subscripting is supposed to take constant time (intuitively as well as per the requirements of the Collection protocol), and looking up the n^{th} Character is impossible without looking at all bytes that come before it.

String.Index, the index type used by String and its views, is an opaque value that essentially stores a byte offset from the beginning of the string. It's still an $O(n)$ operation if you want to compute the index for the n^{th} character and have to start at the beginning of the string, but once you have a valid index, subscripting the string with it now only takes $O(1)$ time. And crucially, finding the next index after an existing index is also fast because you can start at the existing index's byte offset — you don't need to go back to the beginning again. This is why iterating over the characters in a string in order (forward or backward) is efficient.

String index manipulation is based on the same Collection APIs you'd use with any other collection. It's easy to miss this equivalence since the collections we use by far the most — arrays — use integer indices, and we usually use simple arithmetic to manipulate those. The index(after:) method returns the index of the next character:

```
let s = "abcdef"
let second = s.index(after: s.startIndex)
s[second] // b
```

You can automate iterating over multiple characters in one go via the index(_:offsetBy:) method:

```
// Advance 4 more characters
let sixth = s.index(second, offsetBy: 4)
s[sixth] // f
```

If there's a risk of advancing past the end of the string, you can add a limitedBy: parameter. The method returns nil if it hits the limit before reaching the target index:

```
let safeIdx = s.index(s.startIndex, offsetBy: 400, limitedBy: s.endIndex)
safeIdx // nil
```

This is undoubtedly more code than simple integer indices would require, but again, that's the point. If Swift allowed integer subscripting of strings, the temptation to accidentally write horribly inefficient code (e.g. by using integer subscripting inside a loop) would be too big.

Nevertheless, to someone used to dealing with fixed-width characters, working with strings in Swift seems challenging at first — how will you navigate without integer indices? And indeed, some seemingly simple tasks like extracting the first four characters of a string can turn into monstrosities like this one:

```
s[..<s.index(s.startIndex, offsetBy: 4)] // abcd
```

But thankfully, being able to access the string via the Collection interface also means you have several helpful techniques at your disposal. Many of the methods that operate on Array also work on String. Using the prefix method, the same thing looks much clearer:

```
s.prefix(4) // abcd
```

(Note that either expression returns a Substring; you can convert it back into a String by wrapping it in a String.init. We'll talk more about substrings in the next section.)

Iterating over characters in a string is easy without integer indices; just use a for loop. If you want to number each character in turn, use enumerated():

```
for (i, c) in s.enumerated() {
    print("\(i):\(c)")
}
```

Or say you want to find a specific character. In that case, you can use index(of:):

```
var hello = "Hello!"
if let idx = hello.index(of: "!") {
    hello.insert(contentsOf: ", world", at: idx)
}
hello // Hello, world!
```

The insert(contentsOf:at:) method inserts another collection of the same element type (e.g. Character for strings) before a given index. This doesn't have to be another String; you could insert an array of characters into a string just as easily.

Substrings

Like all collections, String has a specific slice or SubSequence type named Substring. A substring is much like an ArraySlice: it's a view of a base string with different start and end indices. Substrings share the text storage of their base strings. This has the huge benefit that slicing a string is a very cheap operation. Creating the firstWord variable in the following example requires no expensive copies or memory allocations:

```
let sentence = "The quick brown fox jumped over the lazy dog."
let firstSpace = sentence.index(of: " ") ?? sentence.endIndex
let firstWord = sentence[..<firstSpace] // The
type(of: firstWord) // Substring
```

Slicing being cheap is especially important in loops where you iterate over the entire (potentially long) string to extract its components. Tasks like finding all occurrences of a word in a text or parsing a CSV file come to mind. A very useful string processing operation in this context is splitting. The split method is defined on Collection and returns an array of subsequences (i.e. [Substring]). Its most common variant is defined like so:

```
extension Collection where Element: Equatable {
    public func split(separator: Element, maxSplits: Int = Int.max,
        omittingEmptySubsequences: Bool = true) -> [SubSequence]
}
```

You can use it like this:

```
let poem = """
Over the wintry
forest, winds howl in rage
```

```
with no leaves to blow.
"""
let lines = poem.split(separator: "\n")
// ["Over the wintry", "forest, winds howl in rage", "with no leaves to blow."]
type(of: lines) // Array<Substring>
```

This can serve a function similar to the components(separatedBy:) method String inherits from NSString, with added configurations for whether or not to drop empty components. Again, no copies of the input string are made. And since there's another variant of split that takes a closure, it can do more than just compare characters. Here's an example of a primitive word wrap algorithm, where the closure captures a count of the length of the line thus far:

```
extension String {
    func wrapped(after: Int = 70) -> String {
        var i = 0
        let lines = self.split(omittingEmptySubsequences: false) {
            character in
            switch character {
            case "\n", " " where i >= after:
                i = 0
                return true
            default:
                i += 1
                return false
            }
        }
        return lines.joined(separator: "\n")
    }
}

sentence.wrapped(after: 15)
/*
The quick brown
fox jumped over
the lazy dog.
*/
```

Or, consider writing a version that takes a sequence of multiple separators:

```
extension Collection where Element: Equatable {
    func split<S: Sequence>(separators: S) -> [SubSequence]
        where Element == S.Element
    {
        return split { separators.contains($0) }
    }
}
```

This way, you can write the following:

```
"Hello, world!".split(separators: ",! ") // ["Hello", "world"]
```

StringProtocol

Substring has almost the same interface as String. This is achieved through a common protocol named StringProtocol, which both types conform to. Since almost the entire string API is defined on StringProtocol, you can mostly work with a Substring as you would with a String. At some point, though, you'll have to turn your substrings back into String instances; like all slices, substrings are only intended for short-term storage, in order to avoid expensive copies during an operation. When the operation is complete and you want to store the results or pass them on to another subsystem, you should create a new String. You can do this by initializing a String with a Substring, as we do in this example:

```
func lastWord(in input: String) -> String? {
    // Process the input, working on substrings
    let words = input.split(separators: [",", " "])
    guard let lastWord = words.last else { return nil }
    // Convert to String for return
    return String(lastWord)
}

lastWord(in: "one, two, three, four, five") // Optional("five")
```

The rationale for discouraging long-term storage of substrings is that a substring always holds on to the entire original string. A substring representing a single character of a huge string will hold the entire string in memory, even after the original string's lifetime would normally have ended. Long-term storage of substrings would therefore effectively cause memory leaks because the original strings have to be kept in memory even when they're no longer accessible.

By working with substrings during an operation and only creating new strings at the end, we defer copies until the last moment and make sure to only incur the cost of those copies that are actually necessary. In the example above, we split the entire (potentially long) string into substrings, but only pay the cost for a single copy of one short substring at the end. (Ignore for a moment that this algorithm isn't efficient anyway; iterating backward from the end until we find the first separator would be the better approach.)

Encountering a function that only accepts a Substring when you want to pass a String is less common — most functions should either take a String or any StringProtocol-conforming type. But if you do need to pass a String, the quickest way is to subscript the string with the range operator ... without specifying any bounds:

```
// Substring with identical start and end index as the base string
let substring = sentence[...]
```

We already saw an example of this in the chapter on collection-protocols when we defined the Words collection.

> *The Substring type is new in Swift 4. In Swift 3, String.CharacterView used to be its own slice type. This had the advantage that users only had to understand a single type, but it meant that a stored substring would keep the entire original string buffer alive even after it normally would've been*

> *released. Swift 4 trades a small loss in convenience for cheap slicing operations and predictable memory usage.*
>
> *The Swift team does recognize that requiring explicit conversions from* Substring *to* String *is a little annoying. If this turns out to be a big problem in practical use, the team is considering wiring an [implicit subtype relationship]() between* Substring *and* String *directly into the compiler, in the same way that* Int *is a subtype of* Optional<Int>*. This would allow you to pass a* Substring *anywhere a* String *is expected, and the compiler would perform the conversion for you.*

You may be tempted to take full advantage of the existence of StringProtocol and convert all your APIs to take StringProtocol instances rather than plain Strings. But the advice of the Swift team is [not to do that]():

> *Our general advice is to stick with* String*. Most APIs would be simpler and clearer just using* String *rather than being made generic (which itself can come at a cost), and user conversion on the way in on the few occasions that's needed isn't much of a burden.*

APIs that are extremely likely to be used with substrings, and at the same time aren't further generalizable to the Sequence or Collection level, are an exception to this rule. An example of this in the standard library is the joined method. Swift 4 added an overload for sequences with StringProtocol-conforming elements:

```
extension Sequence where Element: StringProtocol {
    /// Returns a new string by concatenating the elements of the sequence,
    /// adding the given separator between each element.
    public func joined(separator: String = "") -> String
}
```

This lets you call joined directly on an array of substrings (which you got from a call to split, for example) without having to map over the array and copy every substring into a new string. This is more convenient and much faster.

The number type initializers that take a string and convert it into a number also take StringProtocol values in Swift 4. Again, this is especially handy if you want to process an array of substrings:

```
let commaSeparatedNumbers = "1,2,3,4,5"
let numbers = commaSeparatedNumbers
    .split(separator: ",").flatMap { Int($0) }
// [1, 2, 3, 4, 5]
```

Since substrings are intended to be short-lived, it's generally not advisable to return one from a function unless we're talking about Sequence or Collection APIs that return slices. If you write a similar function that only makes sense for strings, having it return a substring tells readers that it doesn't make a copy. Functions that create new strings requiring memory allocations, such as uppercased(), should always return String instances.

If you want to extend String with new functionality, placing the extension on StringProtocol is a good idea to keep the API surface between String and Substring consistent. StringProtocol is explicitly designed to be used whenever you would've previously extended String. If you want to move existing extensions from String to StringProtocol, the only change you should have to make is to replace any passing of self into an API that takes a concrete String with String(self).

Keep in mind, though, that as of Swift 4, StringProtocol is not yet intended as a conformance target for your own custom string types. The documentation explicitly warns against it:

> *Do not declare new conformances to StringProtocol. Only the String and Substring types of the standard library are valid*

conforming types.

Allowing developers to write their own string types (with special storage or performance optimizations, for instance) is the eventual goal, but the protocol design hasn't yet been finalized, so adopting it now may break your code in Swift 5.

Code Unit Views

Sometimes it's necessary to drop down to a lower level of abstraction and operate directly on Unicode scalars or code units instead of grapheme clusters. String provides three views for this: `unicodeScalars`, `utf16`, and `utf8`. Like `String`, these are bidirectional collections that support all the familiar operations. And like substrings, the views share the string's storage; they simply represent the underlying bytes in a different way.

There are a few common reasons why you'd need to work on one of the views. Firstly, maybe you actually need the code units, perhaps for rendering into a UTF-8-encoded webpage, or for interoperating with a non-Swift API that expects a particular encoding. Or maybe you need information about the string in a specific format.

For example, suppose you're writing a Twitter client. While the Twitter API expects strings to be UTF-8-encoded, Twitter's character counting algorithm is based on NFC-normalized code points (i.e. scalars). So if you want to show your users how many characters they have left, this is how you could do it:

```
let tweet = "Having ☕ in a cafe\u{301} in 🇫🇷 and enjoying the ☀."
let characterCount = tweet
    .precomposedStringWithCanonicalMapping
    .unicodeScalars.count
// 46
```

(NFC normalization converts base letters along with combining marks, such as the e plus accent in "cafe\u{301}", into their precomposed form. The precomposedStringWithCanonicalMapping property is defined in Foundation.)

UTF-8 is the de facto standard for storing text or sending it over the network. Since the utf8 view is a collection, you can use it to pass the raw UTF-8 bytes to any other API that accepts a sequence of bytes, such as the initializers for Data or Array:

```
let utf8Bytes = Data(tweet.utf8)
utf8Bytes.count // 62
```

Note that the utf8 collection doesn't include a trailing null byte. If you need a null-terminated representation, use the withCString method or the utf8CString property on String. The latter returns an array of bytes:

```
let nullTerminatedUTF8 = tweet.utf8CString
nullTerminatedUTF8.count // 63
```

The utf16 view has a special significance because Foundation APIs traditionally treat strings as collections of UTF-16 code units. While the NSString interface is transparently bridged to Swift.String, thereby handling the transformations implicitly for you, other Foundation APIs such as NSRegularExpression or NSAttributedString often expect input in terms of UTF-16 data. We'll see an example of this in the next section.

A second reason for using the code unit views is that operating on code units rather than fully composed characters can be faster. This is because the Unicode grapheme breaking algorithm is fairly complex and requires additional lookahead to identify the start of the next grapheme cluster. Traversing the String as a Character collection got much faster between Swift 3.0 and 4.0, however, so be sure to measure if the (relatively small) speedup is worth your losing Unicode correctness. As soon as you drop down to one of the code unit views, you must be certain that your specific algorithm works correctly on that basis. For example, using the UTF-16 view to parse JSON should be alright because all special characters the parser is interested in (such as commas, quotes, or braces) are representable in a single code unit; it doesn't matter that some strings in the JSON data may contain complex emoji sequences. On the other hand, finding all occurrences of a word in a string wouldn't work correctly on a code unit view if you want the search algorithm to find different normalization forms of the search string. To see just how big the speed difference can be, take a look at the performance section at the end of this chapter.

No Random Access

One desirable feature *none* of the views provides is random access. This is a change from Swift 3, where `String.UTF16View` view did conform to `RandomAccessCollection` (although only when you imported Foundation). This was possible for just this view type because `String` uses UTF-16 or ASCII internally for its in-memory representation. This means the n^{th} UTF-16 code unit would always be at the n^{th} position in the buffer (even if the string was in "ASCII buffer mode" – it's just a question of the width of the entries to advance over). And although this is still true in Swift 4.0, the internal format of `String` is supposed to be an implementation detail. The Swift team wants to keep the door open for adding different backing store types in the future.

The consequence is that `String` and its views are a bad match for algorithms that require random access. The vast majority of string-processing tasks should work fine with sequential traversal, especially since an algorithm can always store substrings for fragments it wants to be able to revisit in constant time. If you absolutely need random access, you can always convert the UTF-8 or UTF-16 view into an array and work on that, as in Array(str.utf16) (again, at the price of sacrificing Unicode correctness).

Index Sharing

Strings and their views share the same index type, String.Index. This means you can use an index derived from the string to subscript one of the views. In the following example, we search the string for "é" (which consists of two scalars, the letter e and the combining accent). The resulting index refers to the first scalar in the Unicode scalar view:

```
let pokemon = "Poke\u{301}mon" // Pokémon
if let index = pokemon.index(of: "é") {
    let scalar = pokemon.unicodeScalars[index] // e
    String(scalar) // e
}
```

This works great as long as you go down the abstraction ladder, from characters to scalars to UTF-16 or UTF-8 code units. Going the other way can fail because not every valid index in one of the code unit views is on a Character boundary. In the following example, accessing the string with an index on a UTF-16 code unit boundary would trap:

```
let family = "□□□□"
// This initializer creates an index at a UTF-16 offset
let someUTF16Index = String.Index(encodedOffset: 2)
//family[someUTF16Index] // trap, invalid index
```

String.Index has a set of methods (samePosition(in:)) and failable initializers (String.Index.init?(_:within:)) for converting indices between views. These return nil if the given index has no exact corresponding position in the specified view. For example, trying to convert the position of the combining accent in the scalars view to a valid index in the string fails because the combining character doesn't have its own position in the string:

```
if let accentIndex = pokemon.unicodeScalars.index(of: "\u{301}") {
    accentIndex.samePosition(in: pokemon) // nil
}
```

Note that there's a bug in Swift 4.0 that sometimes causes these conversions to wrongly succeed when working with complex emoji sequences, such as the family emoji we use above. noCharacterBoundary isn't a valid position in the string, yet the return value is non-nil:

```
let noCharacterBoundary = family.utf16.index(family.utf16.startIndex,
    offsetBy: 3)
// Not a valid index for the character view
noCharacterBoundary.encodedOffset // 3

// Wrong! if let should fail because source index was not on character boundary
if let idx = String.Index(noCharacterBoundary, within: family) {
    // Subscripting returns incomplete Character, this shouldn't happen
    family[idx] // □□□
}
```

Until this is fixed, a reliable way to find the start of Character boundary is to use the Foundation method rangeOfComposedCharacterSequence:

```
extension String.Index {
    func samePositionOnCharacterBoundary(in str: String) -> String.Index {
        let range = str.rangeOfComposedCharacterSequence(at: self)
        return range.lowerBound
    }
}

let validIndex =
    noCharacterBoundary.samePositionOnCharacterBoundary(in: family)
// Works
family[validIndex] // □□□□
```

Strings and Foundation

Swift's String type has a very close relationship with its Foundation counterpart, NSString. Any String instance can be bridged to NSString using the as operator, and Objective-C APIs that take or return an NSString are automatically translated to use String. But that's not all. As of Swift 4.0, String still lacks a lot of functionality that NSString possesses. Since strings are such fundamental types and constantly having to cast to NSString would be annoying, String receives special treatment from the compiler: when you import Foundation, NSString members become directly accessible on String instances, making Swift strings significantly more capable than they'd otherwise be.

Having the additional features is undoubtedly a good thing, but it can make working with strings somewhat confusing. For one thing, if you forget to import Foundation, you may wonder why some methods aren't available. Foundation's history as an Objective-C framework also tends to make the NSString APIs feel a little bit out of place next to the standard library, if only because of different naming conventions. Last but not least, overlap between the feature sets of the two libraries sometimes means there are two APIs with completely different names that perform nearly identical tasks. If you're a long-time Cocoa developer and learned the NSString API before Swift came along, this is probably not a big deal, but it will be puzzling for newcomers.

We've already seen one example — the standard library's split method vs. components(separatedBy:) in Foundation — and there are numerous other mismatches: Foundation uses ComparisonResult enums for comparison predicates, while the standard library is designed around boolean predicates; methods like trimmingCharacters(in:) and components(separatedBy:) take a CharacterSet as an argument, which is an unfortuate misnomer in Swift (more on that later); the extremely powerful enumerateSubstrings(in:options:_:) method, which can iterate over a string in chunks of grapheme clusters, words, sentences, or paragraphs, deals with strings and ranges where a corresponding standard library API would use substrings. (The standard library could also expose the same functionality as a lazy sequence, which would be very cool.)

The following example enumerates the words in a string. The callback closure is called once for every word found:

```
let sentence = """
The quick brown fox jumped \
over the lazy dog.
"""
var words: [String] = []
sentence.enumerateSubstrings(in: sentence.startIndex..., options: .byWords)
{ (word, range, _, _) in
    guard let word = word else { return }
    words.append(word)
}
words
// ["The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]
```

To get an overview of all NSString members imported into String, check out the file NSStringAPI.swift in the Swift source code repository.

Other String-Related Foundation APIs

Having said all that, native NSString APIs are mostly pleasant to use with Swift strings because most of the bridging work is done for you. Many other Foundation APIs that deal with strings are a lot trickier to use because Apple hasn't (yet?) written special Swift overlays for them. Consider NSAttributedString, the Foundation class for representing rich text with formatting. To use attributed strings successfully from Swift, you have to be aware of the following:

- There are two classes, NSAttributedString for immutable strings, and NSMutableAttributedString for mutable ones. They have reference semantics, as opposed to the Swift standard for collections to have value semantics.

- Although all NSAttributedString APIs that originally take an NSString now take a Swift.String, the entire API is still based on NSString's concept of a collection of UTF-16 code units.

For example, the attributes(at: Int, effectiveRange: NSRangePointer?) method for querying the formatting attributes at a specific location expects an integer index (measured in UTF-16 units) rather than a String.Index, and the effectiveRange returned via pointer is an NSRange, not a Range<String.Index>. The same is true for ranges you pass to NSMutableAttributedString.addAttribute(_:value:range:).

NSRange is a structure that contains two integer fields, location and length:

```
public struct NSRange {
    public var location: Int
    public var length: Int
}
```

In the context of strings, the fields specify a string segment in terms of UTF-16 code units. Swift 4 makes working with NSRange somewhat easier than in earlier versions because there are now initializers for converting between Range<String.Index> and NSRange; this doesn't make the extra code required to translate back and forth any shorter, however. Here's an example of how you'd create and modify an attributed string:

```
let text = "□ Click here for more info."
let linkTarget =
    URL(string: "https://www.youtube.com/watch?v=DLzxrzFCyOs")!

// Object is mutable despite ≤ t (reference semantics)
let formatted = NSMutableAttributedString(string: text)

// Modify attributes of a part of the text
if let linkRange = formatted.string.range(of: "Click here") {
    // Convert Swift range to NSRange
    // Note that the start of the range is 3 because the preceding emoji
    // character doesn't fit in a single UTF-16 code unit
    let nsRange = NSRange(linkRange, in: formatted.string) // {3, 10}
    // Add the attribute
    formatted.addAttribute(.link, value: linkTarget, range: nsRange)
}
```

This code adds a link to the "Click here" segment of the string.

Querying the attributed string for the formatting attributes at a specific character position works like this (including the cumbersome process of allocating a pointer to receive the effective range of the returned attributes as an out-parameter because Objective-C has no tuple return types):

```
// Query attributes at start of the word "here"
if let queryRange = formatted.string.range(of: "here"),
    // Get the character index in the UTF-16 view
    let utf16Index = String.Index(queryRange.lowerBound,
        within: formatted.string.utf16)
{
    // Convert index to UTF-16 integer offset
    let utf16Offset = utf16Index.encodedOffset
    // Prepare NSRangePointer to receive effective attributes range
    var attributesRange = UnsafeMutablePointer<NSRange>.allocate(capacity: 1)
    defer {
        attributesRange.deinitialize(count: 1)
        attributesRange.deallocate(capacity: 1)
    }

    // Execute query
    let attributes = formatted.attributes(at: utf16Offset,
        effectiveRange: attributesRange)
    attributesRange.pointee // {3, 10}
    // Convert NSRange back to Range<String.Index>
    if let effectiveRange = Range(attributesRange.pointee, in: formatted.string) {
        // The substring spanned by the attribute
        formatted.string[effectiveRange] // Click here
    }
}
```

We think you'll agree that there's still a long way to go before this code starts to feels like idiomatic Swift.

Aside from NSAttributedString, other Foundation classes with very similar impedance mismatches include NSRegularExpression and the wonderful NSLinguisticTagger.

Ranges of Characters

Speaking of ranges, you may have tried to iterate over a range of characters and found that it doesn't work:

```
let lowercaseLetters = ("a" as Character)..."z"

for c in lowercaseLetters { // Error
    ...
}
```

(The cast to Character is important because the default type of a string literal is String; we need to tell the type checker we want a Character range.)

We talked about the reason why this fails in the chapter about built-in collections: Character doesn't conform to the Strideable protocol, which is a requirement for ranges to become *countable* and thus collections. All you can do with a range of characters is compare other characters against it, i.e. check whether a character is inside the range or not:

```
lowercaseLetters.contains("A") // false
lowercaseLetters.contains("é") // true
```

It's interesting that Swift's default string ordering orders é between e and f, i.e. accented letters are inside the range. Combine that with the fact that a single Character can contain an infinite number of combining marks, and it makes sense that the range can't be countable: it contains an infinite (i.e. uncountable) number of possible characters.

One type for which the notion of countable ranges does make sense is Unicode.Scalar, at least if you stick to ASCII or other well-ordered subsets of the Unicode catalog. Scalars have a well-defined order through their code point values, and there's always a finite number of scalars between any two bounds. Unicode.Scalar isn't Strideable by default, but we can add the conformance retroactively:

```
extension Unicode.Scalar: Strideable {
    public typealias Stride = Int

    public func distance(to other: Unicode.Scalar) -> Int {
        return Int(other.value) - Int(self.value)
    }

    public func advanced(by n: Int) -> Unicode.Scalar {
        return Unicode.Scalar(UInt32(Int(value) + n))!
    }
}
```

(We're ignoring the fact that the surrogate code points 0xD800 to 0xDFFF aren't valid Unicode scalar values. Constructing a range that overlaps this region is a programmer error.)

This allows us to create a countable range of Unicode scalars and use it as a very convenient way to generate an array of characters:

```
let lowercase = ("a" as Unicode.Scalar)..."z"
Array(lowercase.map(Character.init))
/*
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n",
"o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
*/
```

CharacterSet

Let's look at one last interesting Foundation type, and that's CharacterSet. We already mentioned in the built-in collections chapter that this struct should really be called UnicodeScalarSet because that's what it is: an efficient data structure for representing sets of Unicode scalars. It is not at all compatible with the Character type.

We can illustrate this by creating a set from a couple of complex emoji. It seems as though the set only contains the two emoji we put in, but testing for membership with a third emoji is successful because the firefighter emoji is really a sequence of *woman* + ZWJ + *fire engine*:

```
let favoriteEmoji = CharacterSet("□□□□".unicodeScalars)
// Wrong! Or is it?
favoriteEmoji.contains("□") // true
```

CharacterSet provides a number of factory initializers like .alphanumerics or .whitespacesAndNewlines. Most of these correspond to official Unicode character categories (each code point is assigned a category, such as "Letter" or "Nonspacing Mark"). The categories cover all scripts, and not just ASCII or Latin-1, so the number of members in these predefined sets is often huge. The type conforms to SetAlgebra, which defines the interface for set operations such as membership testing and constructing unions or intersections. CharacterSet does *not* conform to Sequence or Collection, so we can't easily count or iterate over all members in a set.

The following example implements an alternative way of splitting a string into words using a CharacterSet via the UnicodeScalarView:

```
extension String {
    func words(with charset: CharacterSet = .alphanumerics) -> [Substring] {
        return self.unicodeScalars.split {
            !charset.contains($0)
        }.map(Substring.init)
    }
}

let code = "struct Array<Element>: Collection { }"
code.words() // ["struct", "Array", "Element", "Collection"]
```

This will break the string apart at every non-alphanumeric character, giving us an array of UnicodeScalarView slices. We then turn them back into substrings via map, passing the substring initializer. The good news is, even after going through this fairly extensive pipeline, the string slices in words will still just be views onto the original string, so this should be more efficient than the components(separatedBy:) method (which returns an array of strings, and thus copies).

Internal Structure of String and Character

Like the other collection types in the standard library, strings are copy-on-write collections with value semantics. A String instance stores a reference to a buffer, which holds the actual character data. When you make a copy of a string (through assignment or by passing it to a function) or create substrings, all these instances share the same buffer. The character data is only copied when an instance gets mutated while it's sharing its character buffer with one or more other instances. For more on copy-on-write, see the chapter on structs and classes.

In Swift 4.0, String uses either 8-bit ASCII (if the string contains nothing but ASCII characters) or UTF-16 (if one or more non-ASCII characters are present) as its in-memory representation. You may be able to use this knowledge to your advantage if you require the best possible performance — traversing the UTF-16 view may be a bit faster than the UTF-8 or Unicode scalar views for non-ASCII data — but keep in mind that the in-memory format is an implementation detail that may change without notice. It's possible that String will become more flexible in the future, e.g. by storing UTF-8-encoded text directly in UTF-8, thereby saving the effort of transcoding it to UTF-16.

Strings received from Objective-C are backed by an NSString. In this case, the NSString acts directly as the Swift string's buffer to make the bridging efficient. An NSString-backed String will be converted to a native Swift string when it gets mutated.

The Character Type

The internal structure of the Character type is interesting. As we've seen, Character represents a sequence of scalars that might be arbitrarily long. At the same time, grapheme clusters of arbitrary length are an edge case — the vast majority of characters are only a few bytes long. It's a good idea to optimize for the common case by storing the bytes that make up the character in line as long as they don't exceed a certain length, and only allocating a separate buffer for uncommonly large grapheme clusters. If you look at the source code, you'll find that Character is essentially defined like this:

```
public struct Character {
    internal enum Representation {
        case smallUTF16(Builtin.Int63)
        case large(Buffer)
    }

    internal var _representation: Representation
}
```

Character is a wrapper around an enum with two cases, .smallUTF16 and .large. The small case is used for grapheme clusters where the UTF-16 representation is 63 bits or less. (Builtin.Int63 is an internal LLVM type that's only available to the standard library.) The unusual size of 63 bits was carefully chosen to fit a Character instance into a single machine word. The remaining bit is needed to discriminate between the two enum cases. The compiler is smart enough to use so-called *extra inhabitants* — bit patterns that aren't valid values for a particular type — in an enum's associated values to store the enum case tag. This works here because the associated value for the large case (which is effectively a pointer) also has some spare bits. Pointer alignment rules mean that some bits of a valid object pointer will always be zero.

```
MemoryLayout<Character>.size // 8
```

This technique — holding a small number of elements internally and switching to a heap-based buffer — is sometimes called the "small string optimization." It works particularly well here since the optimized case is so much more common than the unoptimized one.

String could profit from a similar optimization because many strings used in code are small enough to fit into an 8-byte word. Apple introduced such an optimization for NSString a few years ago, where short strings of up to 7 ASCII-only characters are stored directly in a tagged pointer, saving any extra heap allocations. (There's even an extra mode with custom five- or six-bit encodings for strings of up to 11 characters out of a very constrained alphabet — it's very clever.) Swift strings could get something like this in the future. Check out the Swift core team's String Manifesto to learn more about the plans to provide additional performance enhancements for strings.

A Simple Regular Expression Matcher

To demonstrate some string processing techniques in the contex of a larger example, we'll implement a simple regular expression matcher based on a similar matcher written in C in Brian W. Kernighan and Rob Pike's *The Practice of Programming*. The original code, while beautifully compact, made extensive use of C pointers, so it often doesn't translate well to other languages. But with Swift, through use of optionals and slicing, you can almost match the C version in simplicity.

First, let's define a basic regular expression type:

```
/// A simple regular expression type, supporting ^ and $ anchors,
/// and matching with . and *
public struct Regex {
    private let regexp: String

    /// Construct from a regular expression String
    public init(_ regexp: String) {
        self.regexp = regexp
    }
}
```

Since this regular expression's functionality is going to be so simple, it's not really possible to create an "invalid" regular expression with its initializer. If the expression support were more complex (for example, supporting multi-character matching with []), you'd possibly want to give it a failable initializer.

Next, we extend Regex to add a match function, which takes a string and returns true if it matches the expression:

```
extension Regex {
    /// Returns true if the string argument matches the expression.
    public func match(_ text: String) -> Bool {

        // If the regex starts with ^, then it can only match the
        // start of the input
        if regexp.first == "^" {
            return Regex.matchHere(regexp: regexp.dropFirst(),
                text: text[...])
        }

        // Otherwise, search for a match at every point in the input
        // until one is found
        var idx = text.startIndex
        while true {
            if Regex.matchHere(regexp: regexp[...],
                text: text.suffix(from: idx))
            {
                return true
            }
            guard idx != text.endIndex else { break }
            text.formIndex(after: &idx)
        }

        return false
    }
}
```

The matching function doesn't do much except iterate over every possible substring of the input, from the start to the end, checking if it matches the regular expression from that point on. But if the regular expression starts with a ^, then it need only match from the start of the text.

matchHere is where most of the regular expression-processing logic lies:

```
extension Regex {
    /// Match a regular expression string at the beginning of text.
    private static func matchHere(
        regexp: Substring, text: Substring) -> Bool
    {
        // Empty regexprs match everything
        if regexp.isEmpty {
            return true
        }

        // Any character followed by * requires a call to matchStar
        if let c = regexp.first, regexp.dropFirst().first == "*" {
            return matchStar(character: c, regexp: regexp.dropFirst(2), text: text)
        }

        // If this is the last regex character and it's $, then it's a match iff the
        // remaining text is also empty
        if regexp.first == "$" && regexp.dropFirst().isEmpty {
            return text.isEmpty
        }

        // If one character matches, drop one from the input and the regex
        // and keep matching
        if let tc = text.first, let rc = regexp.first, rc == "." || tc == rc {
            return matchHere(regexp: regexp.dropFirst(), text: text.dropFirst())
        }

        // If none of the above, no match
        return false
    }

    /// Search for zero or more c's at beginning of text, followed by the
    /// remainder of the regular expression.
    private static func matchStar
        (character c: Character, regexp: Substring, text: Substring) -> Bool
    {
        var idx = text.startIndex
        while true { // a * matches zero or more instances
            if matchHere(regexp: regexp, text: text.suffix(from: idx)) {
                return true
            }
            if idx == text.endIndex || (text[idx] != c && c != ".") {
                return false
            }
            text.formIndex(after: &idx)
        }
    }
}
```

The matcher is very simple to use:

```
Regex("^h..lo*!$").match("hellooooo!") // true
```

This code makes extensive use of slicing (both with range-based subscripts and with the dropFirst method) and optionals — especially the ability to equate an optional with a non-optional value. So, for example, if regexp.first == "^" will work, even with an empty string, because "".first returns nil. However, you can still equate that to the non-optional "^", and when it's nil, it'll return false.

The ugliest part of the code is probably the while true loop. The requirement is that this loops over every possible substring, *including* an empty string at the end. This is to ensure that expressions like Regex("$").match("abc") return true. If strings worked similarly to arrays, with an integer index, we could write something like this:

```
// ... means up to _and including_ the endIndex
for idx in text.startIndex...text.endIndex {
    // Slice string between idx and the end
    if Regex.matchHere(regexp: _regexp, text: text[idx...]) {
        return true
    }
}
```

The final time around the for, idx would equal text.endIndex, so text[idx...] would be an empty string.

So why doesn't the for loop work? We mentioned in the built-in collection chapter that ranges are neither sequences nor collections by default. So we can't iterate over a range of string indices because the range isn't a sequence. And we can't use the character view's indices collection either, because it doesn't include its endIndex. As a result, we're stuck with using the C-style while loop.

ExpressibleByStringLiteral

Throughout this chapter, we've been using String("blah") and "blah" pretty much interchangeably, but they're different. "" is a string literal, just like the array literals covered in the collection protocols chapter. You can make your types initializable from a string literal by conforming to ExpressibleByStringLiteral.

String literals are slightly more complicated than array literals because they're part of a hierarchy of three protocols: ExpressibleByStringLiteral, ExpressibleByExtendedGraphemeClusterLiteral, and ExpressibleByUnicodeScalarLiteral. Each defines an init for creating a type from each kind of literal, but unless you really need fine-grained logic based on whether or not the value is being created from a single scalar/cluster, you only need to implement the string version; the others are covered by default implementations that refer to the string literal initializer:

```
extension Regex: ExpressibleByStringLiteral {
    public init(stringLiteral value: String) {
        regexp = value
    }
}
```

Once defined, you can begin using string literals to create the regex matcher by explicitly naming the type:

```
let r: Regex = "^h..lo*!$"
```

Or even better is when the type is already named for you, because the compiler can then infer it:

```
func findMatches(in strings: [String], regex: Regex) -> [String] {
    return strings.filter { regex.match($0) }
}
```

```
findMatches(in: ["foo","bar","baz"], regex: "^b..") // ["bar", "baz"]
```

CustomStringConvertible and CustomDebugStringConvertible

Functions like print, String(describing:), and string interpolation are written to take any type, no matter what. Even without any customization, the results you get back might be acceptable because structs print their properties by default:

```
print(Regex("colou?r"))
// prints out Regex(regexp: "colou?r")
```

Then again, you might want something a little prettier, especially if your type contains private variables you don't want displayed. But never fear! It only takes a minute to give your custom class a nicely formatted output when it's passed to print:

```
extension Regex: CustomStringConvertible {
    public var description: String {
        return "/\(regexp)/"
    }
}
```

Now, if someone converts your custom type to a string through various means — using it with a streaming function like print, passing it to String(describing:), or using it in some string interpolation — it'll print out as /expression/:

```
let regex = Regex("colou?r")
print(regex) // /colou?r/
```

There's also CustomDebugStringConvertible, which you can implement when someone calls String(reflecting:), in order to provide more debugging output than the pretty-printed version:

```
extension Regex: CustomDebugStringConvertible {
    public var debugDescription: String {
        return "{expression:\(regexp)}"
    }
}
```

String(reflecting:) falls back to using CustomStringConvertible if CustomDebugStringConvertible isn't implemented, and vice versa. So often, CustomDebugStringConvertible isn't worth implementing if your type is simple. However, if your custom type is a container, it's probably polite to conform to CustomDebugStringConvertible in order to print the debug versions of the elements the type contains. So we can extend the FIFOQueue example from the collection protocols chapter:

```
extension FIFOQueue: CustomStringConvertible,
    CustomDebugStringConvertible
{
    public var description: String {
        // Print contained elements using String(describing:), which favors
        // CustomStringConvertible
        let elements = map { String(describing: $0) }.joined(separator: ", ")
        return "[\(elements)]"
    }

    public var debugDescription: String {
        // Print contained elements using String(reflecting:), which favors
        // CustomDebugStringConvertible
        let elements = map { String(reflecting: $0) }.joined(separator: ", ")
        return "FIFOQueue: [\(elements)]"
    }
}
```

Note the word "favors" in the comments there — String(describing:) falls back to CustomDebugStringConvertible if CustomStringConvertible isn't available — so if you do anything out of the ordinary when printing for debug, be sure to implement CustomStringConvertible as well. But if your implementations for description and debugDescription are identical, you can pick either one and omit the other.

By the way, Array always prints out the debug description of its elements, even when invoked via String(describing:). The reason given on the swift-dev mailing list was that an array's description should never be presented to the user, so debugging is the only use case. And an array of empty strings would look wrong without the enclosing quotes, which String.description omits.

Given that conforming to CustomStringConvertible implies that a type has a pretty print output, you may be tempted to write something like the following generic function:

```
func doSomethingAttractive<T: CustomStringConvertible>(with value: T) {
    // Print out something incorporating value, safe in the knowledge
    // it will print out sensibly.
}
```

But you're not supposed to use CustomStringConvertible in this manner. Instead of poking at types to establish whether or not they have a description property, you should use String(describing:) regardless and live with the ugly output if a type doesn't conform to the protocol. This will never fail for any type. And it's a good reason to implement CustomStringConvertible whenever you write more than a very simple type. It only takes a handful of lines.

Text Output Streams

The print and dump functions in the standard library log text to the standard output. How does that work? The default versions of these functions forward to overloads named print(_:to:) and dump(_:to:). The to: argument is the output target; it can be any type that conforms to the TextOutputStream protocol:

```
public func print<Target: TextOutputStream>
    (_ items: Any..., separator: String = " ",
    terminator: String = "\n", to output: inout Target)
```

The standard library maintains an internal text output stream that writes everything that's streamed to it to the standard output. What else can you write to? Well, String is the only type in the standard library that's an output stream:

```
var s = ""
let numbers = [1,2,3,4]
print(numbers, to: &s)
s // [1, 2, 3, 4]
```

This is useful if you want to reroute the output of the print and dump functions into a string. Incidentally, the standard library also harnesses output streams to allow Xcode to capture all stdout logging. Take a look at this global variable declaration in the standard library:

```
public var _playgroundPrintHook: ((String) -> Void)?
```

If this is non-nil, print will use a special output stream that routes everything that's printed both to the standard output *and* to this function. The declaration is even public, so you can use this for your own shenanigans:

```
var printCapture = ""
_playgroundPrintHook = { text in
    printCapture += text
}
print("This is supposed to only go to stdout")
printCapture // This is supposed to only go to stdout
```

But don't rely on it! It's totally undocumented, and we don't know what functionality in Xcode will break when you reassign this.

We can also make our own output streams. The protocol has only one requirement: a write method that takes a string and writes it to the stream. For example, this output stream buffers writes to an array:

```
struct ArrayStream: TextOutputStream {
    var buffer: [String] = []
    mutating func write(_ string: String) {
        buffer.append(string)
    }
}

var stream = ArrayStream()
print("Hello", to: &stream)
print("World", to: &stream)
stream.buffer // ["", "Hello", "\n", "", "World", "\n"]
```

The documentation explicitly allows functions that write to an output stream to call write(_:) multiple times per writing operation. That's why the array buffer in the example above contains separate elements for line breaks and even some empty strings. This is an implementation detail of the print function that may change in future releases.

Another possibility is to extend Data so that it takes a stream, writing it as UTF-8-encoded output:

```
extension Data: TextOutputStream {
    mutating public func write(_ string: String) {
        self.append(contentsOf: string.utf8)
    }
}

var utf8Data = Data()
var string = "café"
utf8Data.write(string) // ()
```

The source of an output stream can be any type that conforms to the TextOutputStreamable protocol. This protocol requires a generic method, write(to:), which accepts any type that conforms to TextOutputStream and writes self to it.

In the standard library, String, Substring, Character, and Unicode.Scalar conform to TextOutputStreamable, but you can also add conformance to your own types. One way to do this is with print(_:to:). However, it's very easy to make a mistake here by accidentally forgetting the to: parameter. Unless you test with a target that's not the standard output, you might not even notice the oversight. Another option is to call the target stream's write method directly. This is how the queue type we built in the chapter on collection protocols could adopt TextOutputStreamable:

```
extension FIFOQueue: TextOutputStreamable {
    func write<Target: TextOutputStream>(to target: inout Target) {
        target.write("[")
        target.write(map { String(describing: $0) }.joined(separator: ","))
        target.write("]")
    }
}

var textRepresentation = ""
let queue: FIFOQueue = [1,2,3]
queue.write(to: &textRepresentation)
textRepresentation // [1,2,3]
```

This isn't very different from saying let textRepresentation = String(describing: queue), though, aside from being more complicated. One interesting aspect of output streams is that a source can call write multiple times and the stream will process each write immediately. You can see this if you write the following rather silly sample:

```
struct SlowStreamer: TextOutputStreamable, ExpressibleByArrayLiteral {
    let contents: [String]

    init(arrayLiteral elements: String...) {
        contents = elements
    }

    func write<Target: TextOutputStream>(to target: inout Target) {
        for x in contents {
            target.write(x)
            target.write("\n")
            sleep(1)
        }
    }
}
```

```
let slow: SlowStreamer = [
    "You'll see that this gets",
    "written slowly line by line",
    "to the standard output",
]
print(slow)
```

As new lines are printed to target, the output appears; it doesn't wait for the call to complete.

As we've seen, internally, print is using some TextOutputStream-conforming wrapper on the standard output. You could write something similar for standard error, like this:

```
struct StdErr: TextOutputStream {
    mutating func write(_ string: String) {
        guard !string.isEmpty else { return }

        // Strings can be passed directly into C functions that take a
        // const char* - see the interoperability chapter for more!
        fputs(string, stderr)
    }
}

var standarderror = StdErr()
print("oops!", to: &standarderror)
```

Streams can also hold state, they can potentially transform their output, and you can chain them together. The following output stream replaces all occurrences of the specified phrases with the given alternatives. Like String, it also conforms to TextOutputStreamable, making it both a target and a source of text-streaming operations:

```
struct ReplacingStream: TextOutputStream, TextOutputStreamable {
    let toReplace: DictionaryLiteral<String, String>
    private var output = ""

    init(replacing toReplace: DictionaryLiteral<String, String>) {
        self.toReplace = toReplace
    }

    mutating func write(_ string: String) {
        let toWrite = toReplace.reduce(string) { partialResult, pair in
            partialResult.replacingOccurrences(of: pair.key, with: pair.value)
        }
        print(toWrite, terminator: "", to: &output)
    }

    func write<Target: TextOutputStream>(to target: inout Target) {
        output.write(to: &target)
    }
}

var replacer = ReplacingStream(replacing: [
    "in the cloud": "on someone else's computer"
])

let source = "People find it convenient to store their data in the cloud."
print(source, terminator: "", to: &replacer)

var output = ""
print(replacer, terminator: "", to: &output)
output
// People find it convenient to store their data on someone else's computer.
```

DictionaryLiteral is used in the above code instead of a regular dictionary. This is useful when you want to be able to use the [key: value] literal syntax, but you don't want the two side effects you'd get from using a Dictionary: elimination of duplicate keys and reordering of the keys. If this indeed isn't what you want, then DictionaryLiteral is a nice alternative to an array of pairs (i.e. [(key, value)]) while allowing the caller to use the more convenient [:] syntax.

String Performance

There's no denying that coalescing multiple variable-length UTF-16 values into extended grapheme clusters is going to be more expensive than just ripping through a buffer of 16-bit values. But what's the cost? One way to test performance would be to adapt the regular expression matcher above to work against all of the different string views.

However, this presents a problem. Ideally, you'd write a generic regex matcher with a placeholder for the view. But this doesn't work — String and its views don't all implement a common "string view" protocol. Also, in our regex matcher, we need to represent specific character constants like * and ^ to compare against the regex. In the UTF16View, these would need to be UInt16, but with the string itself, they'd need to be Character. Finally, we want the regex matcher initializer itself to still take a String. How would it know which method to call to get the appropriate view out?

One technique is to bundle up all the variable logic into a single type and then parameterize the regex matcher on that type. First, we define a protocol that has all the necessary information:

```
protocol StringViewSelector {
    associatedtype View: Collection

    static var caret: View.Element { get }
    static var asterisk: View.Element { get }
    static var period: View.Element { get }
    static var dollar: View.Element { get }

    static func view(from s: String) -> View
}
```

This information includes an associated type for the view we're going to use, getters for the four constants needed, and a function to extract the relevant view from a string.

Given this, you can implement concrete versions like so:

```
struct UTF8ViewSelector: StringViewSelector {
    static var caret: UInt8 { return UInt8(ascii: "^") }
    static var asterisk: UInt8 { return UInt8(ascii: "*") }
    static var period: UInt8 { return UInt8(ascii: ".") }
    static var dollar: UInt8 { return UInt8(ascii: "$") }
    static func view(from s: String) -> String.UTF8View { return s.utf8 }
}

struct CharacterViewSelector: StringViewSelector {
    static var caret: Character { return "^" }
    static var asterisk: Character { return "*" }
    static var period: Character { return "." }
    static var dollar: Character { return "$" }

    static func view(from s: String) -> String { return s }
}
```

You can probably guess what UTF16ViewSelector and UnicodeScalarViewSelector look like.

These are what some people call "phantom types" — types that only exist at compile time and don't actually hold any data. Try calling MemoryLayout<CharacterViewSelector>.size — it'll return zero because it contains no data. All we're using these types for is to parameterize behavior of another type: the regex matcher. It'll use them like so:

```
struct Regex<V: StringViewSelector>
    where V.View.Element: Equatable,
    V.View.SubSequence: Collection
{
    let regexp: String
    /// Construct from a regular expression String.
    init(_ regexp: String) {
        self.regexp = regexp
    }
}

extension Regex {
    /// Returns true if the string argument matches the expression.
    func match(_ text: String) -> Bool {
        let text = V.view(from: text)
        let regexp = V.view(from: self.regexp)

        // If the regex starts with ^, then it can only match the start
        // of the input.
        if regexp.first == V.caret {
            return Regex.matchHere(regexp: regexp.dropFirst(), text: text[...])
        }

        // Otherwise, search for a match at every point in the input until
        // one is found.
        var idx = text.startIndex
        while true {
            if Regex.matchHere(regexp: regexp[...], text: text.suffix(from: idx)) {
                return true
            }
            guard idx != text.endIndex else { break }
            text.formIndex(after: &idx)
        }
        return false
    }

    /// Match a regular expression string at the beginning of text.
    private static func matchHere(
        regexp: V.View.SubSequence, text: V.View.SubSequence) -> Bool
    {
        // ...
    }
    // ...
}
```

Once the code is rewritten like this, we can write some benchmarking code that measures the time taken to process some arbitrary regular expression across a very large input:

```
func benchmark<V: StringViewSelector>(_: V.Type, pattern: String, text: String)
    -> TimeInterval
    where V.View.Element: Equatable, V.View.SubSequence: Collection
{
    let r = Regex<V>(pattern)
    let lines = text.split(separator: "\n").map(String.init)
    var count = 0

    let startTime = CFAbsoluteTimeGetCurrent()
    for line in lines {
        if r.match(line) { count = count &+ 1 }
    }
    let totalTime = CFAbsoluteTimeGetCurrent() - startTime
    return totalTime
}

let timeCharacter = benchmark(CharacterViewSelector.self,
    pattern: pattern, text: input)
let timeUnicodeScalar = benchmark(UnicodeScalarViewSelector.self,
    pattern: pattern, text: input)
let timeUTF16 = benchmark(UTF8ViewSelector.self,
    pattern: pattern, text: input)
let timeUTF8 = benchmark(UTF16ViewSelector.self,
    let pattern: pattern, text: input)
```

The results show the following speeds for the different views in processing the regex on a large corpus of English (128,000 lines, 6.5 million characters) and Chinese (43,000 lines, 1.5 million characters) text:

View	ASCII text	Chinese text
Characters	1.4 seconds	3.5 seconds
Unicode.Scalars	1.6 seconds	1.1 seconds
UTF-16	0.7 seconds	0.5 seconds
UTF-8	1.1 seconds	n/a

(Benchmarking the UTF-8 view for Chinese text doesn't make sense because it can deliver incorrect results.)

As you can see, the UTF-16 view is consistently the fastest in this test. But how much faster it is depends on the input. It's also worth noting that processing of Characters got much faster in Swift 4.0. When we ran the ASCII benchmark with Swift 3.0 for the previous edition of this book, the character view was more than 10 times slower than any of the other three. The penalty for Unicode correctness is no longer as big in Swift 4.0.

Only you can know if your use case justifies choosing your view type based on performance. It's almost certainly the case that these performance characteristics only matter when you're doing extremely heavy string manipulation, but if you're certain that what you're doing would be correct when operating on UTF-16, Unicode scalar, or UTF-8 data, this can still give you a decent speedup.

Recap

Strings in Swift are very different than their counterparts in almost all other mainstream programming languages. When you're used to strings effectively being arrays of code units, it'll take a while to switch your mindset to Swift's approach of prioritizing Unicode correctness over simplicity.

Ultimately, we think Swift makes the right choice. Unicode text is much more complicated than what those other languages pretend it is. In the long run, the time savings from avoided bugs you'd otherwise have written will probably outweigh the time it takes to unlearn integer indexing.

We're so used to random "character" access that we may not realize how rarely this feature is really needed in string processing code. We hope the examples in this chapter convince you that simple in-order traversal is perfectly fine for most common operations. Forcing you to be explicit about which representation of a string you want to work on — grapheme clusters, Unicode scalars, UTF-16 or UTF-8 code units — is another safety measure; readers of your code will be grateful for it.

When Chris Lattner outlined the goals for Swift's string implementation in July 2016, he ended with this:

> *Our goal is to be better at string processing than Perl!*

Swift 4 isn't quite there yet — too many desirable features are missing, including moving more string APIs from Foundation into the standard library, native language support for regular expressions, string formatting and parsing APIs, and more powerful string interpolation. The good news it that the Swift team has expressed interest in tackling all these topics in the future.

Error Handling

Swift provides multiple ways for dealing with errors and even allows us to build our own mechanisms. In the chapter on optionals, we looked at two of them: optionals and assertions. An optional indicates that a value may or may not be there; we have to inspect the optional and unwrap the value before we can use it. An assertion validates that a condition is true; if the condition doesn't hold, the program crashes.

Looking at the interfaces of types in the standard library can give us a good feel for when to use an optional and when not to. Optionals are used for operations that have a clear and commonly used "not found" or "invalid input" case. For instance, consider the failable initializer for `Int` that takes a string: it returns `nil` if the input isn't a valid integer. Another example: when you look up a key in a dictionary, it's often expected that the key might not be present. Therefore, a dictionary lookup returns an optional result.

Contrast this with arrays: when looking up an element at a specific index, the element is returned directly and isn't wrapped in an optional. This is because the programmer is expected to know if an array index is valid. Accessing an array with an index that's out of bounds is considered a programmer error, and consequently, this will crash your application. If you're unsure whether or not an index is within bounds, you need to check beforehand.

Assertions are a great tool for identifying bugs in your code. Used correctly, they show you at the earliest possible moment when your program is in a state you didn't expect. They should never be used to signal *expected errors* such as a network error.

Note that arrays also have accessors that return optionals. For example, the `first` and `last` properties on `Collection` return `nil` when called on an empty collection. The developers of Swift's standard library deliberately designed the API this way because it's common to access these values in situations when the collection might be empty.

An alternative to returning an optional from a function that can fail is to mark the function as `throws`. Besides a different syntax for how the caller must handle the success and failure cases, the key difference to returning an optional is that throwing functions can return a rich error value that carries information about the error that occurred.

This difference is a good guideline for when to use one approach over the other. Consider the `first` and `last` properties on `Collection` again. They have exactly one error condition (the collection is empty) — returning a rich error wouldn't give the caller more information than what's already present in the optional value. Compare this to a function that executes a network request: many things can fail, from the network being down, to being unable to parse the server's response. Rich error information is necessary to allow the caller to react differently to different errors (or just to show the user what exactly went wrong).

The Result Type

Before we look at Swift's built-in error handling in more detail, let's discuss the `Result` type, which will help clarify how Swift's error handling works when you take away the syntactic sugar. A

Result type is very similar to an optional. Recall that an optional is just an enumeration with two cases: a .none or nil case, with no associated value; and a .some case, which has an associated value. The Result type is an enum with two cases as well: a failure case, which carries an associated error value; and a success case, also with an associated value. Just like optionals, Result has one generic parameter:

```
enum Result<A> {
    case failure(Error)
    case success(A)
}
```

The failure case constrains its associated value to the Error protocol. We'll come back to this shortly.

Let's suppose we're writing a function to read a file from disk. As a first try, we could define the interface using an optional. Because reading a file might fail, we want to be able to return nil:

```
func contentsOrNil(ofFile filename: String) -> String?
```

The interface above is very simple, but it doesn't tell us anything about why reading the file failed. Does the file not exist? Or do we not have the right permissions? This is another example where the failure reason matters. Let's define an enum for the possible error cases:

```
enum FileError: Error {
    case fileDoesNotExist
    case noPermission
}
```

Now we can change the type of our function to return either an error or a valid result:

```
func contents(ofFile filename: String) -> Result<String>
```

The caller of the function can look at the result cases and react differently based on the error. In the code below, we try to read the file, and in case reading succeeds, we print the contents. If the file doesn't exist, we print that, and we handle any remaining errors in a different way:

```
let result = contents(ofFile: "input.txt")
switch result {
case let .success(contents):
    print(contents)
case let .failure(error):
    if let fileError = error as? FileError,
        fileError == .fileDoesNotExist
    {
        print("File not found")
    } else {
        // Handle error
    }
}
```

Throwing and Catching

Swift's built-in error handling is implemented almost like in the above example, only with a different syntax. Instead of giving a function a Result return type to indicate that it can fail, we mark it as throws. Note that Result applies to types, whereas throws applies to functions (we'll come back to this difference later in this chapter). For every throwing function, the compiler will verify that the caller either catches the error or propagates it to its caller. In the case of contents(ofFile:), the function signature including throws looks like this:

```
func contents(ofFile filename: String) throws -> String
```

From now on, our code won't compile unless we annotate every call to contents(ofFile:) with try. The try keyword serves two purposes: first, it signals to the compiler that we know we're calling a function that can throw an error. Second, and more importantly, it immediately makes clear to readers of the code which functions can throw.

Calling a throwing function also forces us to make a decision about how we want to deal with possible errors. We can either handle an error by using do/catch, or we can have it propagate up the call stack by marking the calling function as throws. We can use pattern matching to catch specific errors or catch all errors. In the example below, we explicitly catch a fileDoesNotExist case and then handle all other errors in a catch-all case. Within the catch-all case, the compiler automatically makes a variable, error, available (much like the implicit newValue variable in a property's willSet handler):

```
do {
    let result = try contents(ofFile: "input.txt")
    print(result)
} catch FileError.fileDoesNotExist {
    print("File not found")
} catch {
    print(error)
    // Handle any other error
}
```

The error handling syntax in Swift probably looks familiar to you. Many other languages use the same try*,* catch*, and* throw *keywords for exception handling. Despite the resemblance, error handling in Swift doesn't incur the runtime cost that's often associated with exceptions. The compiler treats* throw *like a regular return, making both code paths very fast.*

If we want to expose more information in our errors, we can use an enum with associated values. For example, a file parser could choose to model the possible errors like this:

```
enum ParseError: Error {
    case wrongEncoding
    case warning(line: Int, message: String)
}
```

Note that we could've also used a struct or class instead; any type that conforms to the Error protocol can be used as an error in a throwing function. And because the Error protocol has no requirements, any type can choose to conform to it without extra implementation work.

Now, if we want to parse a string, we can again use pattern matching to distinguish between the cases. In the case of .warning, we can bind the line number and warning message to a variable:

```
do {
    let result = try parse(text: "{\"message\":\"We come in peace\"}")
    print(result)
} catch ParseError.wrongEncoding {
    print("Wrong encoding")
} catch let ParseError.warning(line, message) {
    print("Warning at line \(line): \(message)")
} catch {
}
```

Something about the above code doesn't feel quite right. Even if we're absolutely sure that the only error that could happen is of type ParseError (which we handled exhaustively), we still need to write a final catch case to convince the compiler that we caught all possible errors. In a future Swift version, the compiler might be able to check exhaustiveness within a module, but across modules, this problem can't be solved. The reason is that Swift errors are untyped: we can only mark a function as throws, but we can't specify *which* errors it'll throw. This was a deliberate design decision — most of the time, you only care about whether or not an error was present. If we needed to specify the types of all errors, this could quickly get out of hand: it would make functions' type signatures quite complicated, especially for functions that call several other throwing functions and propagate their errors. Moreover, adding a new error case would be a breaking change for all clients of the API.

That said, Swift might get typed errors someday; this topic is being actively discussed on the mailing lists. If typed errors come to Swift, we expect them to be an opt-in feature because untyped errors are still the better choice in many situations. For example, frameworks like Cocoa probably won't ever have typed errors.

> *Because errors are untyped, it's important to document the types of errors your functions can throw. Xcode supports a Throws keyword in documentation markup for this purpose. Here's an example:*
>
> ```
> /// Opens a text file and returns its contents.
> ///
> /// - Parameter filename: The name of the file to read.
> /// - Returns: The file contents, interpreted as UTF-8.
> /// - Throws: `FileError` if the file does not exist or
> /// the process doesn't have read permissions.
> func contents(ofFile filename: String) throws -> String
> ```
>
> *The Quick Help popover that appears when you Option-click on the function name will now include an extra section for the thrown errors.*

Typed Errors

It can sometimes be useful to take advantage of the type system to specify the concrete errors a function can throw. We can come up with a slightly altered Result type, which has an additional generic parameter for the error type:

```
enum Result<A, ErrorType: Error> {
    case failure(ErrorType)
    case success(A)
```

```
}
```

This way, we can declare functions using an explicit error type. The following parseFile function will return either an array of strings or a ParseError. We don't have to handle any other cases when calling it, and the compiler knows this:

```
func parse(text: String) -> Result<[String], ParseError>
```

In code where your errors have a significant semantic meaning, you can choose to use a typed Result type instead of Swift's built-in error handling. This way, you can let the compiler verify that you caught all possible errors. However, in most applications, using throws and do/try/catch will lead to simpler code. There's another big benefit to using the built-in error handling: the compiler will make sure you can't ignore the error case when calling a function that might throw. With the definition of parseFile above, we could write the following code:

```
_ = parse(text: invalidData)
```

If the function were marked as throws, the compiler would force us to call it using try. The compiler would also force us to either wrap that call in a do/catch block or propagate the error.

Granted, the example above is unrealistic, because ignoring the parse function's return value doesn't make any sense, and the compiler *would* force you to consider the failure case when you unwrap the result. It's relevant when dealing with functions that don't have a meaningful return value, though, and in these situations, it can really help you not forget to catch the error. For example, consider the following function:

```
func setupServerConnection() throws
```

Because the function is marked as throws, we need to call it with try. If the server connection fails, we probably want to switch to a different code path or display an error. By having to use try, we're forced to think about this case. Had we chosen to return a Result<()> instead, it would be all too easy to ignore errors.

Bridging Errors to Objective-C

Objective-C has no mechanism that's similar to throws and try. (Objective-C *does* have exception handling that uses these same keywords, but exceptions in Objective-C should only be used to signal programmer errors. You rarely catch an Objective-C exception in a normal app.)

Instead, the common pattern in Cocoa is that a method returns NO or nil when an error occurs. In addition, failable methods take a reference to an NSError pointer as an extra argument; they can use this pointer to pass concrete error information to the caller. For example, the contents(ofFile:) method would look like this in Objective-C:

```
- (NSString *)contentsOfFile(NSString *)filename error:(NSError **)error;
```

Swift automatically translates methods that follow this pattern to the throws syntax. The error parameter gets removed since it's no longer needed, and BOOL return types are changed to Void. This is helpful when dealing with existing frameworks in Objective-C. The method above gets imported like this:

```
func contents(ofFile filename: String) throws -> String
```

Other NSError parameters — for example, in asynchronous APIs that pass an error back to the caller in a completion block — are bridged to the Error protocol, so you don't generally need to interact with NSError directly.

If you pass a pure Swift error to an Objective-C method, it'll be bridged to NSError. Since all NSError objects must have a domain string and an integer error code, the runtime will generate default values, using the qualified type name as the domain and numbering the enum cases from zero for the error code. Optionally, you can provide your own values by conforming your type to the CustomNSError protocol.

For example, we could extend our ParseError like this:

```
extension ParseError: CustomNSError {
    static let errorDomain = "io.objc.parseError"
    var errorCode: Int {
        switch self {
        case .wrongEncoding: return 100
        case .warning(_, _): return 200
        }
    }
    var errorUserInfo: [String: Any] {
        return [:]
    }
}
```

In a similar manner, you can add conformance to one or both of the following protocols to make your errors more meaningful and provide better interoperability with Cocoa conventions:

- **LocalizedError** — provides localized messages describing why the error occurred (failureReason), tips for how to recover (recoverySuggestion), and additional help text (helpAnchor).

- **RecoverableError** — describes an error the user can recover from by presenting one or more recoveryOptions and performing the recovery when the user requests it.

Even without conforming to LocalizedError, every type that conforms to Error has a localizedDescription property. Any type conforming to Error can also define a custom *localizedDescription. However, as it's not a requirement of the Error protocol, the property isn't dynamically dispatched. Unless you also conform to LocalizedError, your custom localizedDescription won't be used by Objective-C APIs or if you're inside an Error existential container. See the protocols chapter for more information on dynamic dispatch and existential containers.*

Errors and Function Parameters

In the following example, we'll write a function that checks a list of files for validity. The checkFile function can return three possible values. If it returns true, the file is valid. If it returns false, the file is invalid. If it throws an error, something went wrong when checking the file:

```
func checkFile(filename: String) throws -> Bool
```

As a first step, we can write a simple function that loops over a list of filenames and makes sure that checkFile returns true for every file. If checkFile returns false, we want to make sure to exit early, avoiding any unnecessary work. Since we don't catch any errors that checkFile throws, the first error would propagate to the caller and thus also exit early:

```
func checkAllFiles(filenames: [String]) throws -> Bool {
    for filename in filenames {
        guard try checkFile(filename: filename) else { return false }
    }
    return true
}
```

Checking whether all the elements in an array conform to a certain condition is something that might happen more often in our app. For example, consider the function checkPrimes, which checks whether all numbers in a given list are prime numbers. It works in exactly the same way as checkAllFiles. It loops over the array and checks all elements for a condition (isPrime), exiting early when one of the numbers doesn't satisfy the condition:

```
func checkPrimes(_ numbers: [Int]) -> Bool {
    for number in numbers {
        guard number.isPrime else { return false }
    }
    return true
}

checkPrimes([2,3,7,17]) // true
checkPrimes([2,3,4,5]) // false
```

Both functions mix the process of iterating over a sequence (the for loops) with the actual logic that decides if an element meets the condition. This is a good opportunity to create an abstraction for this pattern, similar to map or filter. To do that, we can add a function named all(matching:) to Sequence. Like filter, all takes a function that performs the condition check as an argument. The difference to filter is the return type. all returns true if all elements in the sequence satisfy the condition, whereas filter returns the elements themselves:

```
extension Sequence {
    /// Returns `true` iff all elements satisfy the predicate
    func all(matching predicate: (Element) -> Bool) -> Bool {
        for element in self {
            guard predicate(element) else { return false }
        }
        return true
    }
}
```

This allows us to rewrite checkPrimes in a single line, which makes it easier to read once you know what all does, and it helps us focus on the essential parts:

```
func checkPrimes2(_ numbers: [Int]) -> Bool {
    return numbers.all { $0.isPrime }
}
```

However, we can't rewrite checkAllFiles to use all, because checkFile is marked as throws. We could easily rewrite all to accept a throwing function, but then we'd have to change checkPrimes too, either by annotating checkPrimes as throwing, by using try!, or by wrapping the call to all in a do/catch block. Alternatively, we could define two versions of all: one that throws and one that doesn't. Except for the try call, their implementations would be identical.

Rethrows

Fortunately, there's a better way. By marking all as rethrows, we can write both variants in one go. Annotating a function with rethrows tells the compiler that this function will only throw an error when its function parameter throws an error. This allows the compiler to waive the requirement that all must be called with try when the caller passes in a non-throwing check function:

```
extension Sequence {
    func all(matching predicate: (Element) throws -> Bool) rethrows
        -> Bool {
        for element in self {
            guard try predicate(element) else { return false }
        }
        return true
    }
}
```

The implementation of checkAllFiles is now very similar to checkPrimes, but because the call to all can now throw an error, we need to insert an additional try:

```
func checkAllFiles(filenames: [String]) throws -> Bool {
    return try filenames.all(matching: checkFile)
}
```

Almost all sequence and collection functions in the standard library that take a function argument are annotated with rethrows. For example, the map function is only throwing if the transformation function is a throwing function itself.

Cleaning Up Using defer

Let's go back to the contents(ofFile:) function from the beginning of this chapter for a minute and have a look at the implementation. In many languages, it's common to have a try/finally construct, where the block marked with finally is always executed when the function returns, regardless of whether or not an error was thrown. The defer keyword in Swift has a similar purpose but works a bit differently. Like finally, a defer block is always executed when a scope is exited, regardless of the reason of exiting — whether it's because a value is successfully returned, because an error happened, or any other reason. Unlike finally, a defer block doesn't require a leading try or do block, and it's more flexible in terms of where you place it in your code:

```
func contents(ofFile filename: String) throws -> String
{
    let file = open("test.txt", O_RDONLY)
    defer { close(file) }
    let contents = try process(file: file)
    return contents
}
```

While defer is often used together with error handling, it can be useful in other contexts too — for example, when you want to keep the code for initialization and cleanup of a resource close together. Putting related parts of the code close to each other can make your code significantly more readable, especially in longer functions.

If there are multiple defer blocks in the same scope, they're executed in reverse order; you can think of them as a stack. At first, it might feel strange that the defer blocks run in reverse order. However, if we look at an example, it should quickly make sense:

```
guard let database = openDatabase(...) else { return }
defer { closeDatabase(database) }
guard let connection = openConnection(database) else { return }
defer { closeConnection(connection) }
guard let result = runQuery(connection, ...) else { return }
```

If an error occurs — for example, during the runQuery call — we want to close the connection first and the database second. Because the defer is executed in reverse order, this happens automatically. The runQuery depends on openConnection, which in turn depends on openDatabase. Therefore, cleaning these resources up needs to happen in reverse order.

There are some situations in which defer blocks don't get executed: when your program segfaults, or when it raises a fatal error (e.g. using fatalError or by force-unwrapping a nil), all execution halts immediately.

Errors and Optionals

Errors and optionals are both very common ways for functions to signal that something went wrong. Earlier in this chapter, we gave you some advice on how to decide which pattern you should use for your own functions. You'll end up working a lot with both errors and optionals, and passing results to other APIs will often make it necessary to convert back and forth between throwing functions and optional values.

The try? keyword allows us to ignore the error of a throws function and convert the return value into an optional that tells us if the function succeeded or not:

```
if let result = try? parse(text: input) {
    print(result)
}
```

Using the try? keyword means we receive less information than before: we only know if the function returned a successful value or if it returned some error — any specific information about that error gets thrown away. To go the other way, from an optional to a function that throws, we have to provide the error value that gets used in case the optional is nil. Here's an extension on Optional that, given an error, does this:

```
extension Optional {
    /// Unwraps `self` if it is non-`nil`.
    /// Throws the given error if `self` is `nil`.
    func or(error: Error) throws -> Wrapped {
        switch self {
            case let x?: return x
            case nil: throw error
        }
    }
}
do {
    let int = try Int("42").or(error: ReadIntError.couldNotRead)
} catch {
    print(error)
}
```

This can be useful in conjunction with multiple try statements, or when you're working inside a function that's already marked as throws.

The existence of the try? keyword may appear contradictory to Swift's philosophy that ignoring errors shouldn't be allowed. However, you still have to explicitly write try? so that the compiler forces you to acknowledge your actions. In cases where you're not interested in the error message, this can be very helpful.

It's also possible to write equivalent functions for converting between Result and throws, or between throws and Result, or between optionals and Result.

There's a third variant of try: try!. This is used when you know there can't possibly be an error result. Just like force-unwrapping an optional value that's nil, try! crashes when the result was actually an error.

Chaining Errors

Chaining multiple calls to functions that can throw errors becomes trivial with Swift's built-in error handling — there's no need for nested if statements or similar constructs; we simply place these calls into a single do/catch block (or wrap them in a throwing function). The first error that occurs breaks the chain and switches control to the catch block (or propagates the error to the caller):

```
func checkFilesAndFetchProcessID(filenames: [String]) -> Int {
    do {
        try filenames.all(matching: checkFile)
        let pidString = try contents(ofFile: "Pidfile")
        return try Int(pidString).or(error: ReadIntError.couldNotRead)
    } catch {
        return 42 // Default value
    }
}
```

Chaining Result

To see how well Swift's native error handling matches up against other error handling schemes, let's compare this to an equivalent example based on the Result type. Chaining multiple functions that return a Result — where the input to the second function is the result of the first — is a lot of work if you want to do it manually. To do so, you call the first function and unwrap its return value; if it's a .success, you can pass the wrapped value to the second function and start over. As soon as one function returns a .failure, the chain breaks and you short-circuit by immediately returning the failure to the caller.

To refactor this, we should turn the common steps of unwrapping the Result, short-circuiting in case of failure, and passing the value to the next transformation in case of success, into a separate function. That function is the flatMap operation. Its structure is identical to the existing flatMap for optionals that we covered in the optionals chapter:

```
extension Result {
    func flatMap<B>(transform: (A) -> Result<B>) -> Result<B> {
        switch self {
        case let .failure(m): return .failure(m)
        case let .success(x): return transform(x)
        }
    }
}
```

With this in place, the end result is quite elegant:

```
func checkFilesAndFetchProcessID(filenames: [String]) -> Result<Int> {
    return filenames
        .all(matching: checkFile)
        .flatMap { _ in contents(ofFile: "Pidfile") }
        .flatMap { contents in
            Int(contents).map(Result.success)
            ?? .failure(ReadIntError.couldNotRead)
    }
}
```

(We're using variants of all(matching:), checkFile, and contents(ofFile:) here that return Result values. The implementations aren't shown here.)

But you can also see that Swift's error handling stands up extremely well. It's shorter, and it's arguably more readable and easier to understand.

Higher-Order Functions and Errors

One domain where Swift's error handling unfortunately does *not* work very well is asynchronous APIs that need to pass errors to the caller in callback functions. Let's look at a function that asynchronously computes a large number and calls back our code when the computation has finished:

```
func compute(callback: (Int) -> ())
```

We can call it by providing a callback function. The callback receives the result as the only parameter:

```
compute { result in
    print(result)
}
```

If the computation can fail, we could specify that the callback receives an optional integer, which would be nil in case of a failure:

```
func computeOptional(callback: (Int?) -> ())
```

Now, in our callback, we must check whether the optional is non-nil, e.g. by using the ?? operator:

```
computeOptional { result in
    print(result ?? -1)
}
```

But what if we want to report specific errors to the callback, rather than just an optional? This function signature seems like a natural solution:

```
func computeThrows(callback: (Int) throws -> ())
```

But this type has a totally different meaning. Instead of saying that the computation might fail, it expresses that the callback itself could throw an error. This highlights the key difference we mentioned earlier: optionals and Result work on types, whereas throws works only on function types. Annotating a function with throws means that the *function* might fail.

It becomes a bit clearer when we try to rewrite the wrong attempt from above using Result:

```
func computeResult(callback: (Int) -> Result<()>)
```

This isn't correct either. We need to wrap the Int argument in a Result, not the callback's return type. Finally, this is the correct solution:

```
func computeResult(callback: (Result<Int>) -> ())
```

Unfortunately, there's currently no clear way to write the variant above with throws. The best we can do is wrap the Int inside another throwing function. This makes the type more complicated:

```
func compute(callback: (() throws -> Int) -> ())
```

And using this variant becomes more difficult for the caller too. In order to get the integer out, the callback now has to call the throwing function. This is where the caller must perform the error checking:

```
compute { (resultFunc: () throws -> Int) in
    do {
        let result = try resultFunc()
        print(result)
    } catch {
        print("An error occurred: \(error)")
    }
}
```

This works, but it's definitely not idiomatic Swift; Result is the way to go for asynchronous error handling. It's unfortunate that this creates an impedance mismatch with synchronous functions that use throws. The Swift team has expressed interest in extending the throws model to other scenarios, but this will likely be part of the much greater task of adding native concurrency features to the language, and that won't happen until Swift 5 at the earliest.

Until then, we're stuck with using our own custom Result types. Apple did consider adding a Result type to the standard library, but ultimately decided against it on the grounds that it wasn't independently valuable enough outside the error handling domain and that the team didn't want to endorse it as an alternative to throws-style error handling. Luckily, using Result for asynchronous APIs is a pretty well-established practice among the Swift developer community, so you shouldn't hesitate to use it in your APIs when the native error handling isn't appropriate. It certainly beats the Objective-C style of having completion handlers with two nullable arguments (a result object and an error object).

Recap

When Apple introduced its error handling model in Swift 2.0, a lot of people in the community were skeptical. People were rolling their own `Result` types in the Swift 1.x days, mostly with a typed error case. The fact that `throws` uses untyped errors was seen as a needless deviation from the strict typing in other parts of the language. Unsurprisingly, the Swift team had considered this very carefully and intentionally went for untyped errors. We were skeptical too, but in hindsight, we think the Swift team was proven correct, not least by the large acceptance of the error handling model in the developer community.

There's a chance that strongly typed error handling will be added as an opt-in feature in the future, along with better support for asynchronous errors and passing errors around as values. As it stands now, error handling is a good example of Swift being a pragmatic language that optimizes for the most common use case first. Keeping the syntax familiar for developers who are used to C-style languages is a more important goal than adhering to the "purer" functional style based on `Result` and `flatMap`.

In the meantime, we now have many possible choices for handling the unexpected in our code. When we can't possibly continue, we can use `fatalError` or an assertion. When we're not interested in the kind of error, or if there's only one kind of error, we can use optionals. When we need more than one kind of error or want to provide additional information, we can use Swift's built-in errors or write our own `Result` type. When we want to write functions that take a function as a parameter, `rethrows` lets us write one variant for both throwing and non-throwing function parameters. Finally, the `defer` statement is very useful in combination with the built-in errors. `defer` statements provide us with a single place to put our cleanup code, regardless of how a scope exits normally or with an error.

Generics

Like most modern languages, Swift has a number of features that can all be grouped under *generic programming*. Generic code allows you to write reusable functions and data types that can work with any type that matches the constraints you define. For example, types such as `Array` and `Set` are generic over their elements. Generic functions are generic over their input and/or output types. The declaration `func identity<A>(input: A) -> A` defines a function that works on any type, identified by the placeholder `A`. In a way, we can also think of protocols with an associated type as "generic protocols." The associated type allows us to abstract over specific implementations. `IteratorProtocol` is an example of such a protocol: it's generic over the `Element` type it produces.

The goal of generic programming is to express the *essential interface* an algorithm or data structure requires. For example, consider the `last(where:)` method we wrote in the built-in collections chapter. Writing this method as an extension on `Array` would've been the obvious choice, but an `Array` has lots of capabilities `last(where:)` doesn't need. By asking ourselves what the essential interface is — that is, the minimal set of features required to implement the desired functionality — we can make the function available to a much broader set of types. In this example, `last(where:)` has only one requirement: it needs to traverse a sequence of elements in reverse order. This makes an extension on `Sequence` the right place for this algorithm (and we can add a more efficient variant to `BidirectionalCollection`).

In this chapter, we'll look at how to write generic code. We'll start by talking about overloading because this concept is closely related to generics. We'll use generic programming to provide multiple implementations of an algorithm, each relying on a different set of assumptions. We'll also discuss some common difficulties you may encounter when writing generic algorithms for collections. We'll then look at how you can use generic data types to refactor code to make it testable and more flexible. Finally, we'll cover how the compiler handles generic code and what we can do to get the best performance for our own generic code.

Overloading

Overloaded functions, i.e. multiple functions that have the same name but different argument and/or return types, are not generic per se. But like generics, they allow us to make one interface available to multiple types.

Overload Resolution for Free Functions

For example, we could define a function named raise(_:to:) to perform exponentiation and provide separate overloads for Double and Float arguments:

```
func raise(_ base: Double, to exponent: Double) -> Double {
    return pow(base, exponent)
}

func raise(_ base: Float, to exponent: Float) -> Float {
    return powf(base, exponent)
}
```

We used the pow and powf functions declared in Swift's Darwin module (or Glibc on Linux) for the implementations.

When the raise function is called, the compiler picks the correct overload based on the types of the arguments and/or the return value:

```
let double = raise(2.0, to: 3.0) // 8.0
type(of: double) // Double
let float: Float = raise(2.0, to: 3.0) // 8.0
type(of: float) // Float
```

Swift has a complex set of rules for which overloaded function to pick, which is based on whether or not a function is generic and what kind of type is being passed in. While the rules are too long to go into here, they can be summarized as "pick the most specific one." This means that non-generic functions are picked over generic ones.

As an example, consider the following function that logs certain properties of a view. We provide a generic implementation for UIView that logs the view's class name and frame, and a specific overload for UILabel that prints the label's text:

```
func log<View: UIView>(_ view: View) {
    print("It's a \(type(of: view)), frame: \(view.frame)")
}

func log(_ view: UILabel) {
    let text = view.text ?? "(empty)"
    print("It's a label, text: \(text)")
}
```

Passing a UILabel will call the label-specific overload, whereas passing any other view will use the generic one:

```
let label = UILabel(frame: CGRect(x: 20, y: 20, width: 200, height: 32))
label.text = "Password"
log(label) // It's a label, text: Password

let button = UIButton(frame: CGRect(x: 0, y: 0, width: 100, height: 50))
```

```
log(button) // It's a UIButton, frame: (0.0, 0.0, 100.0, 50.0)
```

It's important to note that overloads are resolved statically at compile time. This means the compiler bases its decision of which overload to call on the static types of the variables involved, and not on the values' dynamic types at runtime. This is why the generic overload of `log` will be used for both views if we put the label and the button into an array and then call the `log` function in a loop over the array's elements:

```
let views = [label, button] // Type of views is [UIView]
for view in views {
    log(view)
}
/*
It's a UILabel, frame: (20.0, 20.0, 200.0, 32.0)
It's a UIButton, frame: (0.0, 0.0, 100.0, 50.0)
*/
```

The static type of the `view` variable is `UIView`; it's irrelevant that the dynamic type `UILabel` would result in a different overload at runtime.

If instead you need runtime polymorphism — that is, you want the function to be picked based on what a variable points to and not what the type of the variable is — you should be using methods not functions, i.e. define `log` as a method on `UIView` and `UILabel`.

Overload Resolution for Operators

The compiler exhibits some surprising behavior when it comes to the resolution of overloaded operators. As Matt Gallagher points out, the type checker always favors non-generic overloads over generic variants, even when the generic version would be the better choice (and the one which would be chosen if we were talking about a normal function).

Going back to the exponentiation example from above, let's define a custom operator named ** for the same operation:

```
// Exponentiation has higher precedence than multiplication
precedencegroup ExponentiationPrecedence {
    associativity: left
    higherThan: MultiplicationPrecedence
}
infix operator **: ExponentiationPrecedence

func **(lhs: Double, rhs: Double) -> Double {
    return pow(lhs, rhs)
}
func **(lhs: Float, rhs: Float) -> Float {
    return powf(lhs, rhs)
}

2.0 ** 3.0 // 8.0
```

The above code is equivalent to the raise function we implemented in the previous section. Let's add another overload for integers next. We want exponentiation to work for all integer types, so we define a generic overload for all types conforming to BinaryInteger:

```
func **<I: BinaryInteger>(lhs: I, rhs: I) -> I {
    // Cast to Int64, use Double overload to compute result
    let result = Double(Int64(lhs)) ** Double(Int64(rhs))
    return I(result)
}
```

This looks like it should work, but if we now call ** with two integer literals, the compiler complains about an ambiguous use of the ** operator:

```
2 ** 3 // Error: Ambiguous use of operator '**'
```

The explanation for why this happens brings us back to what we said at the beginning of this section: when resolving overloaded operators, the type checker always favors non-generic overloads over generic overloads. Apparently, the compiler ignores the generic overload for integers and then raises the error because it can't decide whether it should call the overload for Double or the one for Float — both are equally valid choices for two integer literal arguments. To convince the compiler to pick the correct overload, we have to explicitly mark at least one of the arguments as an integer type or else provide an explicit result type:

```
let intResult: Int = 2 ** 3 // 8
```

The compiler only behaves in this manner for operators — a generic overload of the raise function for BinaryInteger would work just fine without introducing ambiguities. The reason for this discrepancy comes down to performance: the Swift team considers the reduction of complexity the type checker has to deal with significant enough to warrant the use of this simpler but sometimes incorrect overload resolution model for operators.

Overloading with Generic Constraints

You'll often encounter overloading in conjunction with generic code when the same operation can be expressed with multiple algorithms that each require different constraints on their generic parameters. Suppose we want to determine if all the entries in one array are also contained in another. In other words, we want to find out if the first array is a *subset* of the second (keeping in mind that the element order is inconsequential). The standard library provides a method named isSubset(of:) that does this, but only for types that conform to the SetAlgebra protocol, such as Set.

If we wanted to write a version of isSubset(of:) that works with a broader set of types, it'd look something like this:

```
extension Sequence where Element: Equatable {
    /// Returns true iff all elements in self are also in other.
    func isSubset(of other: [Element]) -> Bool {
        for element in self {
            guard other.contains(element) else {
                return false
            }
        }
        return true
    }
}

let oneToThree = [1,2,3]
let fiveToOne = [5,4,3,2,1]
oneToThree.isSubset(of: fiveToOne) // true
```

isSubset is defined as an extension on the Sequence protocol, where the elements of the sequence are Equatable. It takes an array of the same type to check against, and it returns true if every single element in the receiver is also a member of the argument array. The method doesn't care about what the elements are, just so long as they conform to Equatable, because only equatable types can be used with contains. The element type can be an Int, a String, or your own custom user-defined class, just so long as it's equatable.

This version of isSubset comes with a big downside, which is performance. The algorithm has performance characteristics of $O(nm)$, where n and m are the element counts of the two arrays. That is, as the input sizes grow, the worst-case time the function takes to run grows quadratically. This is because contains runs in linear time on arrays, i.e. $O(m)$. This makes sense if you think about what it needs to do: loop over the contents of the source sequence, checking if they match a given element. But our algorithm calls contains inside another loop — the one over the receiver's elements — which *also* runs in linear time in a similar fashion. And running an $O(m)$ loop inside an $O(n)$ loop results in an $O(nm)$ function.

We can write a better-performing version by tightening the constraints on the sequence's element type. If we require the elements to conform to Hashable, we can convert the other array into a Set, taking advantage of the fact that a set performs lookups in constant time:

```
extension Sequence where Element: Hashable {
    /// Returns true iff all elements in self are also in other.
    func isSubset(of other: [Element]) -> Bool {
        let otherSet = Set(other)
        for element in self {
            guard otherSet.contains(element) else {
                return false
            }
        }
        return true
    }
}
```

With the contains check taking $O(1)$ time (assuming a uniform distribution of the hashes), the entire for loop now becomes $O(n)$ — the time required grows linearly with the number of elements in the receiver. The additional cost of converting other into a set is $O(m)$, but we incur it only once, outside the loop. Thus, the total cost becomes $O(n + m)$, which is much better than the $O(nm)$ for the Equatable version — if both arrays have 1,000 elements, it's the difference between 2,000 and one million iterations.

So now we have two versions of the algorithm, none of which is clearly better than the other — one is faster, and the other works with a wider range of types. The good news is that you don't have to pick one of these options. You can implement both overloads of isSubset and let the compiler select the most appropriate one based on the argument types. Swift is very flexible about overloading — you can overload not just by input type or return type, but also based on different constraints on a generic placeholder, as seen in this example.

The general rule that the type checker will pick the most specific overload it can find also applies here. Both versions of isSubset are generic, so the rule that non-generic variants are favored over generic ones is no help. But the version that requires the elements to be Hashable is more specific, because Hashable extends Equatable, and thus it imposes more constraints. Given that these constraints are probably there to make the algorithm more efficient — as they are in the case of isSubset — the compiler works on the assumption that the more specific function is the better choice.

There's another way in which isSubset can be made more general. Up until now, it's only taken an array of elements to check against. But Array is a specific type. Really, isSubset doesn't need to be this specific. Across the two versions, there are only two function calls: contains in both, and Set.init in the Hashable version. In both cases, these functions only require an input type that conforms to the Sequence protocol:

```
extension Sequence where Element: Equatable {
    /// Returns a Boolean value indicating whether the sequence contains the
    /// given element.
    func contains(_ element: Element) -> Bool
}

struct Set<Element: Hashable>:
    SetAlgebra, Hashable, Collection, ExpressibleByArrayLiteral
{
    /// Creates a new set from a finite sequence of items.
    init<Source: Sequence>(_ sequence: Source)
        where Source.Element == Element
}
```

Given this, the only thing isSubset needs is for other to be of some type that also conforms to Sequence. What's more is that the two sequence types — self and other — *don't have to be the same*. They just need to be sequences of the same element. So here's the Hashable version, rewritten to operate on any two kinds of sequence:

```
extension Sequence where Element: Hashable {
    /// Returns true iff all elements in self are also in other.
    func isSubset<S: Sequence>(of other: S) -> Bool
        where S.Element == Element
    {
        let otherSet = Set(other)
        for element in self {
            guard otherSet.contains(element) else {
                return false
            }
        }
        return true
    }
}
```

Now that the two sequences don't have to be the same type, this opens up a lot more possibilities. For example, you could pass in a CountableRange of numbers to check against:

```
[5,4,3].isSubset(of: 1...10) // true
```

A similar change can be made to the version that requires the elements to be equatable (not shown here).

Parameterizing Behavior with Closures

The isSubset method is still not as general as it could be. What about sequences where the elements aren't equatable? Arrays, for example, aren't equatable, and if we use them as elements, it won't work with our current implementation. Arrays do have an == operator, defined like this:

```
/// Returns true if these arrays contain the same elements.
```

```
func ==<Element: Equatable>(lhs: [Element], rhs: [Element]) -> Bool
```

But that doesn't mean you can use them with isSubset:

```
// Error: Type of expression is ambiguous without more context
[[1,2]].isSubset(of: [[1,2], [3,4]])
```

This is because Array doesn't conform to Equatable. It can't, because the type the array contains might not, itself, be equatable. Swift currently lacks support for *conditional protocol conformance*, i.e. the ability to express the idea that Array (or any Sequence) only conforms to a protocol when certain constraints, such as Element: Equatable, are met. So Array *can* provide an implementation of == for when the contained type is equatable, but it *can't* conform to the protocol.

So how can we make isSubset work with non-equatable types? We can do this by giving control of what equality means to the caller, requiring them to supply a function to determine equality. For example, the standard library provides a second version of contains that does this:

```
extension Sequence {
    /// Returns a Boolean value indicating whether the sequence contains an
    /// element that satisfies the given predicate.
    func contains(where predicate: (Element) throws -> Bool)
        rethrows -> Bool
}
```

That is, it takes a function that takes an element of the sequence and performs some check. It runs the check on each element, returning true as soon as the check returns true. This version of contains is much more powerful. For example, you can use it to check for any condition inside a sequence:

```
let isEven = { $0 % 2 == 0 }
(0..<5).contains(where: isEven) // true
[1, 3, 99].contains(where: isEven) // false
```

We can leverage this more flexible version of contains to write a similarly flexible version of isSubset:

```
extension Sequence {
    func isSubset<S: Sequence>(of other: S,
        by areEquivalent: (Element, S.Element) -> Bool) -> Bool
    {
        for element in self {
            guard other.contains(where: { areEquivalent(element, $0) }) else {
                return false
            }
        }
        return true
    }
}
```

Now, we can use isSubset with arrays of arrays by supplying a closure expression that compares the arrays using ==. Unfortunately, if we import Foundation, another performance optimization for the type checker confuses the compiler so that it no longer knows which version of == to use. So we need a type annotation somewhere in the code:

```
[[1,2]].isSubset(of: [[1,2] as [Int], [3,4]]) { $0 == $1 } // true
```

The two sequences' elements don't even have to be of the same type, so long as the supplied closure handles the comparison:

```
let ints = [1,2]
let strings = ["1","2","3"]
ints.isSubset(of: strings) { String($0) == $1 } // true
```

Operating Generically on Collections

Generic algorithms on collections often pose some special problems, particularly when it comes to working with indices and slices. In this section, we'd like to demonstrate how to deal with these problems by showing two examples that heavily rely on the correct handling of indices and slices.

A Binary Search

Suppose you find yourself in need of a binary search algorithm for collections. You reach for your nearest favorite reference, which happens to be written in Java, and port the code to Swift. Here's one such attempt — albeit a boringly iterative, rather than recursive, one:

```
extension Array {
    /// Returns the first index where value appears in self, or nil,
    /// if value is not found.
    ///
    /// - Requires: areInIncrea sin gOrder is a strict weak ordering over the
    /// elements in self, and the elements in the array are already
    /// ordered by it.
    /// - Complexity: O(log count)
    func binarySearch(for value: Element,
        areInIncreasingOrder: (Element, Element) -> Bool) -> Int?
    {
        var left = 0
        var right = count - 1
        while left <= right {
            let mid = (left + right) / 2
            let candidate = self[mid]
            if areInIncreasingOrder(candidate,value) {
                left = mid + 1
            } else if areInIncreasingOrder(value,candidate) {
                right = mid - 1
            } else {
                // If neither element comes before the other, they _must_ be
                // equal, per the strict ordering requirement of areInIncreasingOrder
                return mid
            }
        }
        // Not found
        return nil
    }
}

extension Array where Element: Comparable {
    func binarySearch(for value: Element) -> Int? {
        return self.binarySearch(for: value, areInIncreasingOrder: <)
    }
}
```

For such a famous and seemingly simple algorithm, a binary search is notoriously hard to get right. This one contains a bug that also existed in the Java implementation for two decades — one we'll fix in the generic version. But we also don't guarantee that it's the only bug!

It's worth noting some of the conventions from the Swift standard library that this follows:

- Similar to index(of:), it returns an optional index, with nil representing "not found."
- It's defined twice — once with a user-supplied parameter to perform the comparison, and once relying on Comparable conformance to supply that parameter as a convenience to callers.
- The ordering must be a strict weak ordering. This means that when comparing two elements, if neither is ordered before the other, they *must* be equal.

This works for arrays, but if you wanted to binary search a ContiguousArray or an ArraySlice, you're out of luck. The method should really be in an extension to RandomAccessCollection — the random access is necessary to preserve the logarithmic complexity, as you need to be able to locate the midpoint in constant time.

A shortcut might be to require the collection to have an Int index. This will cover almost every random-access collection in the standard library, and it means you can cut and paste the entire Array version as is:

```
extension RandomAccessCollection where Index == Int, IndexDistance == Int {
    public func binarySearch(for value: Element,
        areInIncreasingOrder: (Element, Element) -> Bool)
        -> Index?
    {
        // Identical implementation to that of Array...
    }
}
```

Warning: If you do this, you'll introduce an *even worse* bug, which we'll come to shortly.

But this is still restricted to integer-indexed collections, and collections don't always have an integer index. Dictionary, Set, and the various String collection views have custom index types. The most notable random-access example in the standard library is ReversedRandomAccessCollection, which, as we saw in the collection protocols chapter, has an opaque index type that wraps the original index, converting it to the equivalent position in the reversed collection.

Generic Binary Search

If you lift the requirement for an Int index, you'll hit several compiler errors. The code needs some rewrites in order to be fully generic, so here's a fully generic version:

```
extension RandomAccessCollection {
    public func binarySearch(for value: Element,
        areInIncreasingOrder: (Element, Element) -> Bool) -> Index?
    {
        guard !isEmpty else { return nil }
        var left = startIndex
        var right = index(before: endIndex)
        while left <= right {
            let dist = distance(from: left, to: right)
            let mid = index(left, offsetBy: dist/2)
            let candidate = self[mid]
            if areInIncreasingOrder(candidate, value) {
                left = index(after: mid)
            } else if areInIncreasingOrder(value, candidate) {
                right = index(before: mid)
            } else {
                // If neither element comes before the other, they _must_ be
                // equal, per the strict ordering requirement of areInIncreasingOrder
                return mid
            }
        }
        // Not found
        return nil
    }
}

extension RandomAccessCollection where Element: Comparable {
    func binarySearch(for value: Element) -> Index? {
        return binarySearch(for: value, areInIncreasingOrder: <)
    }
}
```

The changes are small but significant. First, the `left` and `right` variables have changed type to no longer be integers. Instead, we're using the start and end index values. These might be integers, but they might also be opaque types like `String`'s index type (or `Dictionary`'s or `Set`'s, not that these are random access).

But secondly, the simple statement `(left + right) / 2` has been replaced by the slightly uglier `index(left, offsetBy: dist/2)`, where `let dist = distance(from: left, to: right)`. How come?

The key concept here is that there are actually two types involved in this calculation: `Index` and `IndexDistance`. These are *not necessarily the same thing*. When using integer indices, we happen to be able to use them interchangeably. But loosening that requirement breaks this.

The distance is the number of times you'd need to call index(after:) to get from one point in the collection to another. The end index must be "reachable" from the start index — there's always a finite integer number of times you need to call index(after:) to get to it. This means IndexDistance must be an integer (though not necessarily an Int), and the corresponding constraint in the definition of Collection is (the constraint is actually defined in _Indexable, and Collection inherits from _Indexable):

```
public protocol Collection: /* ... */ {
    /// A type that can represent the number of steps between a pair of
    /// indices.
    associatedtype IndexDistance: SignedInteger = Int
}
```

This is also why we need an extra guard to ensure the collection isn't empty. When you're just doing integer arithmetic, there's no harm in generating a right value of -1 and then checking that it's less than zero. But when dealing with any kind of index, you need to make sure you don't move back through the start of the collection, which might be an invalid operation. (For example, what would happen if you tried to go back one from the start of a doubly linked list?)

Being integers, index distances can be added together or divided to find the remainder. What we *can't* do is add two indices of any kind together, because what would that mean? If you had the Words collection from the collection protocols chapter, you obviously couldn't "add" two indices together and divide by two. Instead, we must think only in terms of moving indices by some distance using index(after:), index(before:), or index(_:offsetBy:).

This way of thinking takes some getting used to if you're accustomed to thinking in terms of arrays. But think of many array index expressions as a kind of shorthand. For example, when we wrote let right = count - 1, what we really meant was right = index(startIndex, offsetBy: count - 1). It's just that when the index is an Int and startIndex is zero, this reduces to 0 + count - 1, which in turn reduces to count - 1.

This leads us to the serious bug in the implementation that took our Array code and just applied it to RandomAccessCollection: collections with integer indices don't necessarily start with an index of zero, the most common example being ArraySlice. A slice created via myArray[3..<5] will have a startIndex of three. Try and use our simplistic generic binary search on it, and it'll crash *at runtime*. While we were able to require that the index be an integer, the Swift type system has no good way of requiring that the collection be zero-based. And even if it did, in this case, it'd be a silly requirement to impose, since we know a better way. Instead of adding together the left and right indices and halving the result, we find half the distance between the two, and then we advance the left index by that amount to reach the midpoint.

> *This version also fixes the bug in our initial implementation. If you didn't spot it, it's that if the array is extremely large, then adding two integer indices together might overflow before being halved (suppose* count *was approaching* Int.max *and the searched-for element was the last one in the array). On the other hand, when adding half the distance between the two indices, this doesn't happen. Of course, the chances of anyone ever hitting this bug is very low, which is why the bug in the Java standard library took so long to be discovered.*

Now, we can use our binary search algorithm to search ReversedRandomAccessCollection:

```
let a = ["a", "b", "c", "d", "e", "f", "g"]
let r = a.reversed()
r.binarySearch(for: "g", areInIncreasingOrder: >) == r.startIndex // true
```

And we can also search slices, which aren't zero-based:

```
let s = a[2..<5]
s.startIndex // 2
s.binarySearch(for: "d") // Optional(3)
```

Shuffling Collections

To help cement this concept, here's another example, this time an implementation of the Fisher-Yates shuffling algorithm:

```
extension Array {
    mutating func shuffle() {
        for i in 0..<(count - 1) {
            let j = Int(arc4random_uniform(UInt32(count - i))) + i
            self.swapAt(i, j)
        }
    }

    /// Non-mutating variant of shuff ≤
    func shuffled() -> [Element] {
        var clone = self
        clone.shuffle()
        return clone
    }
}
```

Again, we've followed a standard library practice: providing an in-place version, since this can be done more efficiently, and then providing a non-mutating version that generates a shuffled copy of the array, which can be implemented in terms of the in-place version.

So how can we write a generic version of this that doesn't mandate integer indices? Just like with binary search, we still need random access, but we also have a new requirement that the collection be mutable, since we want to be able to provide an in-place version. The use of `count - 1` will definitely need to change in a way similar to the binary search.

Before we get to the generic implementation, there's an extra complication. We want to use arc4random_uniform to generate random numbers, but we don't know exactly what type of integer IndexDistance will be. We know it's an integer, but not necessarily that it's an Int. To handle this, we need to write a generic version of arc4random_uniform that works on any integer type conforming to the BinaryInteger protocol (Swift 4 implemented a new hierarchy of protocol-oriented integers, which makes this task easier than before):

```
extension BinaryInteger {
    static func arc4random_uniform(_ upper_bound: Self) -> Self {
        precondition(
            upper_bound > 0 && UInt32(upper_bound) < UInt32.max,
            "arc4random_uniform only callable up to\(UInt32.max)")
        return Self(Darwin.arc4random_uniform(UInt32(upper_bound)))
    }
}
```

You could write a version of arc4random that operates on ranges spanning negative numbers, or above the max of UInt32, if you wanted to. But to do so would require a lot more code. If you're interested, the definition of arc4random_uniform is actually open source and quite well commented, and it gives several clues as to how you might do this.

We then use the ability to generate a random number for every IndexDistance type in our generic shuffle implementation:

```
extension MutableCollection where Self: RandomAccessCollection {
    mutating func shuffle() {
        var i = startIndex
        let beforeEndIndex = index(before: endIndex)
        while i < beforeEndIndex {
            let dist = distance(from: i, to: endIndex)
            let randomDistance = IndexDistance.arc4random_uniform(dist)
            let j = index(i, offsetBy: randomDistance)
            self.swapAt(i, j)
            formIndex(after: &i)
        }
    }
}

extension Sequence {
    func shuffled() -> [Element] {
        var clone = Array(self)
        clone.shuffle()
        return clone
    }
}

var numbers = Array(1...10)
numbers.shuffle()
numbers // [7, 2, 8, 10, 1, 6, 9, 4, 5, 3]
```

The generic shuffle method is significantly more complex and less readable than the non-generic version. This is partly because we had to replace simple integer math like `count - 1` with index calculations like `index(before: endIndex)`. The other reason is that we switched from a `for` to a `while` loop. The alternative, iterating over the indices with `for i in indices.dropLast()`, has a potential performance problem that we already talked about in the collection protocols chapter: if the `indices` property holds a reference to the collection, mutating the collection while traversing the indices will defeat the copy-on-write optimizations and cause the collection to make an unnecessary copy.

Admittedly, the chances of this happening in our case are small, because most random-access collections likely use plain integer indices where the `Indices` type doesn't need to reference its base collection. For instance, `Array.Indices` is `CountableRange<Int>` instead of the default `DefaultRandomAccessIndices`.

Inside the loop, we measure the distance from the running index to the end and then use our new `BinaryInteger.arc4random_uniform` method to compute a random index to swap with. The actual swap operation remains the same as it is in the non-generic version.

You might wonder why we didn't extend MutableCollection when implementing the non-modifying shuffle. Again, this is a pattern you see often in the standard library — for example, when you sort a ContiguousArray, you get back an Array and not a ContiguousArray.

In this case, the reason is that our immutable version relies on the ability to clone the collection and then shuffle it in place. This, in turn, relies on the collection having value semantics. But not all collections are guaranteed to have value semantics. If NSMutableArray conformed to MutableCollection (which it doesn't — probably because it's bad form for Swift collections to not have value semantics — but could), then shuffled and shuffle would have the same effect, since NSMutableArray has reference semantics. var clone = self just makes a copy of the reference, so a subsequent clone.shuffle would shuffle self — probably not what the user would expect. Instead, we take a full copy of the elements into an array and shuffle and return that.

There's a compromise approach. You could write a version of shuffle to return the same type of collection as the one being shuffled, so long as that type is also a RangeReplaceableCollection:

```
extension MutableCollection
    where Self: RandomAccessCollection, Self: RangeReplaceableCollection
{
    func shuffled() -> Self {
        var clone = Self()
        clone.append(contentsOf: self)
        clone.shuffle()
        return clone
    }
}
```

This relies on the two abilities of RangeReplaceableCollection: to create a fresh empty version of the collection, and to then append any sequence (in this case, self) to that empty collection, thus guaranteeing a full clone takes place. The standard library doesn't take this approach — probably because the consistency of always creating an array for any kind of non-in-place operation is preferred — but it's an option if you want it. However, remember to create the sequence version as well, so that you offer shuffling for non-mutable collections and sequences.

Designing with Generics

As we've seen, generics can be used to provide multiple implementations of the same functionality. We can write generic functions but provide specific implementations for certain types. Also, by using protocol extensions, we can write generic algorithms that operate on many types.

Generics can also be very helpful during the design of your program, in order to factor out shared functionality and reduce boilerplate. In this section, we'll refactor a normal piece of code, pulling out the common functionality by using generics. Not only can we make generic functions, but we can also make generic data types.

Let's write some functions to interact with a web service. For example, consider fetching a list of users and parsing it into the `User` datatype. We write a function named `loadUsers`, which loads the users from the network asynchronously and then calls a callback with the list of fetched users.

We start by implementing it in a naive way. First, we build the URL. Then, we load the data synchronously (this is just for the sake of the example; you should always do networking asynchronously in production code). Next, we parse the JSON response, which gives us an array of dictionaries. Finally, we transform the plain JSON objects into `User` structs:

```
func loadUsers(callback: ([User]?) -> ()) {
    let usersURL = webserviceURL.appendingPathComponent("/users")
    let data = try? Data(contentsOf: usersURL)
    let json = data.flatMap {
        try? JSONSerialization.jsonObject(with: $0, options: [])
    }
    let users = (json as? [Any]).flatMap { jsonObject in
      jsonObject.flatMap(User.init)
    }
    callback(users)
}
```

Notice that we don't take advantage of the Codable system we discussed in the chapter on encoding and decoding here. We use the "old-fashioned" way of parsing JSON into a dictionary with JSONSerialization, and then the User type knows how to initialize itself from a JSON dictionary.

The loadUsers function has three possible error cases: the URL loading can fail, the JSON parsing can fail, and building user objects from the JSON array can fail. All three operations return nil upon failure. By using flatMap on the optional values, we ensure that subsequent operations are only executed if the previous ones succeeded. Otherwise, the nil value from the first failing operation will be propagated through the chain to the end, where we eventually call the callback either with a valid users array or nil.

Now, if we want to write the same function to load other resources, we'd need to duplicate most of the code. For example, if we consider a function to load blog posts, its type could look like this:

```
func loadBlogPosts(callback: ([BlogPost])? -> ())
```

The implementation would be almost the same. Not only would we have duplicated code, but both functions are hard to test: we need to make sure the test can access the web service or else find some way to fake the requests. And because the functions take a callback, we need to make the tests asynchronous too.

Extracting Common Functionality

A better approach is to try to move the User-specific parts outside the function in order to make the other parts reusable. For example, we could add the path and the conversion function that turns JSON objects into domain objects as parameters. We name this new function loadResource to indicate its universality. Because we want the conversion function to handle arbitrary types, we also make the function generic over A:

```
func loadResource<A>(at path: String,
    parse: (Any) -> A?, callback: (A?) -> ())
{
    let resourceURL = webserviceURL.appendingPathComponent(path)
    let data = try? Data(contentsOf: resourceURL)
    let json = data.flatMap {
        try? JSONSerialization.jsonObject(with: $0, options: [])
    }
    callback(json.flatMap(parse))
}
```

Now we can base our loadUsers function on loadResource:

```
func loadUsers(callback: ([User]?) -> ()) {
    loadResource(at: "/users", parse: jsonArray(User.init), callback: callback)
}
```

We use a helper function, jsonArray, to convert JSON into user objects. It first tries to convert an Any to an array of Anys, and then it tries to parse each element using the supplied function, returning nil if any of the steps fail:

```
func jsonArray<A>(_ transform: @escaping (Any) -> A?) -> (Any) -> [A]? {
    return { array in
        guard let array = array as? [Any] else {
            return nil
        }
        return array.flatMap(transform)
    }
}
```

To load the blog posts, we just change the path and the parsing function:

```
func loadBlogPosts(callback: ([BlogPost]?) -> ()) {
    loadResource(at: "/posts", parse: jsonArray(BlogPost.init), callback: callback)
}
```

This avoids a lot of duplication. And if we later decide to switch from synchronous to asynchronous networking, we don't need to update either loadUsers or loadBlogPosts. But even though these functions are now very short, they're still hard to test — they remain asynchronous and depend on the web service being accessible.

Creating a Generic Data Type

The path and parse parameters of the loadResource function are very tightly coupled; if you change one, you likely have to change the other too. Let's bundle them up in a struct that describes a resource. Just like functions, structs (and other types) can also be generic:

```
struct Resource<A> {
    let path: String
    let parse: (Any) -> A?
}
```

Now we can write an alternative version of loadResource as a method on Resource. It uses the resource's properties to determine what to load and how to parse the result, so the only remaining argument is the callback function:

```
extension Resource {
    func loadSynchronously(callback: (A?) -> ()) {
        let resourceURL = webserviceURL.appendingPathComponent(path)
        let data = try? Data(contentsOf: resourceURL)
        let json = data.flatMap {
            try? JSONSerialization.jsonObject(with: $0, options: [])
        }
        callback(json.flatMap(parse))
    }
}
```

The previous top-level functions to load a specific resource now become instances of the Resource struct. This makes it very easy to add new resources without having to create new functions:

```
let usersResource: Resource<[User]> =
    Resource(path: "/users", parse: jsonArray(User.init))
let postsResource: Resource<[BlogPost]> =
    Resource(path: "/posts", parse: jsonArray(BlogPost.init))
```

Adding a variant that uses asynchronous networking is now possible with minimal duplication. We don't need to change any of our existing code describing the endpoints:

```
extension Resource {
    func loadAsynchronously(callback: @escaping (A?) -> ()) {
        let resourceURL = webserviceURL.appendingPathComponent(path)
        let session = URLSession.shared
        session.dataTask(with: resourceURL) { data, response, error in
            let json = data.flatMap {
                try? JSONSerialization.jsonObject(with: $0, options: [])
            }
            callback(json.flatMap(self.parse))
        }.resume()
    }
}
```

Aside from the usage of the asynchronous URLSession API, the only material difference to the synchronous version is that the callback argument must now be annotated with @escaping because the callback function escapes the method's scope. See the functions chapter if you want to learn more about escaping vs. non-escaping closures.

We've now completely decoupled our endpoints from the network calls. We boiled down `usersResource` and `postsResource` to their absolute minimums so that they only describe where the resource is located and how to parse it. The design is also extensible: adding more configuration options, such as the HTTP method or a way to add `POST` data to a request, is simply a matter of adding additional properties to the `Resource` type. (Specifying default values, e.g. `GET` for the HTTP method, helps keep the code clean.)

Testing has become much simpler. The `Resource` struct is fully synchronous and independent of the network, so testing whether a resource is configured correctly becomes trivial. The networking code is still harder to test, of course, because it's naturally asynchronous and depends on the network. But this complexity is now nicely isolated in the `loadAsynchronously` method; all other parts are simple and don't involve asynchronous code.

In this section, we started with a non-generic function for loading some data from the network. Next, we created a generic function with multiple parameters, significantly reducing duplicate code. Finally, we bundled up the parameters into a separate `Resource` data type. The domain-specific logic for the concrete resources is fully decoupled from the networking code. Changing the network stack doesn't require any change to the resources.

How Generics Work

How do generics work from the perspective of the compiler? To answer this question, let's take a closer look at the min function from the standard library (we took this example from the Optimizing Swift Performance session at Apple's 2015 Worldwide Developers Conference):

```
func min<T: Comparable>(_ x: T, _ y: T) -> T {
    return y < x ? y : x
}
```

The only constraints for the arguments and return value of min are that all three must have the same type T and that T must conform to Comparable. Other than that, T could be anything — Int, Float, String, or even a type the compiler knows nothing about at compile time because it's defined in another module. This means that the compiler lacks two essential pieces of information it needs to emit code for the function:

- The sizes of the variables of type T (including the arguments and return value).
- The address of the specific overload of the < function that must be called at runtime.

Swift solves these problems by introducing a level of indirection for generic code. Whenever the compiler encounters a value that has a generic type, it boxes the value in a container. This container has a fixed size to store the value; if the value is too large to fit, Swift allocates it on the heap and stores a reference to it in the container.

The compiler also maintains a list of one or more *witness tables* per generic type parameter: one so-called *value witness table*, plus one *protocol witness table* for each protocol constraint on the type. The witness tables (also called *vtables*) are used to dynamically dispatch function calls to the correct implementations at runtime.

The value witness table is always present for any generic type. It contains pointers to the fundamental operations of the type, such as allocation, copying, and destruction. These can be simple no-ops or memcopies for primitive value types such as Int, whereas reference types include their reference counting logic here. The value witness table also records the size and alignment of the type.

The record for the generic type T in our example will include one protocol witness table because T has one protocol constraint, namely Comparable. For each method or property the protocol declares, the witness table contains a pointer to the implementation for the conforming type. Any calls to one of these methods in the body of the generic function are then dispatched at runtime through the witness table. In our example, the expression y < x is dispatched in this way.

The protocol witness tables provide a mapping between the protocols to which the generic type conforms (this is statically known to the compiler through the generic constraints) and the functions that implement that functionality for the specific type (these are only known at runtime). In fact, the only way to query or manipulate the value in any way is through the witness tables. We couldn't declare the min function with an unconstrained parameter <T> and then expect it to work with any type that has an implementation for <, regardless of Comparable conformance. The compiler wouldn't allow this because there wouldn't be a witness table for it to locate the correct < implementation. This is why generics are so closely related to protocols — you can't do much with unconstrained generics except write container types like Array<Element> or Optional<Wrapped>.

In summary, the code the compiler generates for the min function looks something like this (in pseudocode):

```
func min<T: Comparable>(_ x: Box_T, _ y: Box_T,
    valueWTable_T: VTable, comparableWTable_T: VTable)
    -> Box_T
{
    let xCopy = valueWTable_T.copy(x)
    let yCopy = valueWTable_T.copy(y)
    let result = comparableWTable_T.lessThan(yCopy, xCopy) ? y : x
    valueWTable_T.release(xCopy)
    valueWTable_T.release(yCopy)
    return result
}
```

> *The layout of the container for generic parameters is similar but not identical to the existential containers used for protocol types that we'll cover in the next chapter. An existential container combines the storage for the value and the pointers to zero or more witness tables in one structure, whereas the container for a generic parameter only includes the value storage — the witness tables are stored separately so that they can be shared between all variables of the same type in the generic function.*

If you want to learn more about how the generics system works, Swift compiler developers Slava Pestov and John McCall gave a talk on this topic at the 2017 LLVM Developers' Meeting. We highly recommend that you watch it.

Generic Specialization

The compile-once-and-dispatch-dynamically model we described in the previous section was an important design goal for Swift's generics system. Compared to how C++ templates work — where the compiler generates a separate instance of the templated function or class for every permutation of concrete types using the template — this leads to faster compilation times and potentially smaller binaries. Swift's model is also more flexible because, unlike C++, code that uses a generic API doesn't need to see the implementation of the generic function or type, only the declaration.

The downside is lower performance at runtime, caused by the indirection the code has to go through. This is likely negligible when you consider a single function call, but it does add up when generics are as pervasive as they are in Swift. The standard library uses generics everywhere, including for very common operations that must be as fast as possible, such as comparing values. Even if generic code is only slightly slower than non-generic code, chances are developers will try to avoid using it.

Luckily, the Swift compiler can make use of an optimization called *generic specialization* to remove the overhead. Generic specialization means that the compiler clones a generic type or function, such as `min<T>`, for a concrete parameter type, such as `Int`. This specialized function can then be optimized specifically for `Int`, removing all indirection. So the specialized version of `min<T>` for `Int` would look like this:

```swift
func min(_ x: Int, _ y: Int) -> Int {
    return y < x ? y : x
}
```

This is exactly the implementation you'd write for a concrete, non-generic `min` function. Generic specialization not only eliminates the cost of the virtual dispatch, but it also enables further optimizations, such as inlining, for which the indirection would otherwise be a barrier.

The optimizer uses heuristics to decide which generic types or functions it chooses to specialize, and for which concrete types it performs the specialization. This decision requires a balancing of compilation times, binary size, and runtime performance. If your code calls `min` with `Int` arguments very frequently, but only once with `Float` arguments, chances are only the `Int` variant will be specialized. Make sure to compile your release builds with optimizations enabled (`swiftc -O` on the command line) to take advantage of all available heuristics.

Regardless of how aggressive the compiler is with generic specialization, the generic version of the function will always exist, at least if the generic function is visible to other modules. This ensures that external code can always call the generic version, even if the compiler didn't know anything about the external types when it compiled the generic function.

Whole Module Optimization

Generic specialization only works if the compiler can see the full definition of the generic type or function it wants to specialize. Since Swift compiles source files individually by default, it can only perform specialization if the code that uses the generic code resides in the same file as the generic code.

Since this is a pretty big limitation, the compiler has a flag to enable *whole module optimization*. In this mode, all files in the current module are optimized together as if they were one file, allowing generic specializations across the full codebase. You can enable whole module optimization by passing `-whole-module-optimization` to `swiftc`. Be sure to do so in release builds (and possibly also in debug builds), because the performance gains can be huge. The drawback is longer compilation times.

> *Whole module optimization enables other important optimizations. For example, the optimizer will recognize when an* `internal` *class has no subclasses in the entire module. Since the* `internal` *modifier makes sure the class isn't visible outside the module, this means the compiler can replace dynamic dispatch with static dispatch for all methods of this class.*

Generic specialization requires the definition of the generic type or function to be visible, and therefore can't be performed across module boundaries. This means your generic code will likely be faster for clients that reside in the same module where the code is defined than for external clients. The only exception to this is generic code in the standard library. Because the standard library is so important and used by every other module, definitions in the standard library are visible in all modules and thus available for specialization.

Swift includes an unofficial attribute called @_specialize that allows you to make specialized versions of your generic code available to other modules. You must specify the list of types you want to specialize, so this only helps if you know your code will mostly be used with a limited number of types. Making specialized versions of our min function for integers and strings available to other modules would look like this:

```
@_specialize(exported: true, where T == Int)
@_specialize(exported: true, where T == String)
public func min<T: Comparable>(_ x: T, _ y: T) -> T {
    return y < x ? y : x
}
```

Notice we made the function public — it makes no sense to specify @_specialize for internal, fileprivate, or private APIs, since those are not visible outside the module anyway.

The similar (and equally unsupported) @_inlineable attribute instructs the compiler to make the bodies of annotated functions available to the optimizer when building client code. This effectively removes the cross-module optimization barrier that would otherwise exist. The advantage @_inlineable has over @_specialize is that the originating module doesn't have to hardcode the list of concrete types because the specialization is performed when the client module is compiled.

The standard library uses @_inlineable extensively. The Swift team would like to make it an officially supported attribute (likely with the non-underscored name @inlinable) in the future.

Recap

In the beginning of this chapter, we defined generic programming as identifying the essential interface an algorithm or data type requires. We achieved this for our isSubset method by starting with a non-generic version and then carefully removing constraints. Writing multiple overloads with different constraints allowed us to provide the function to the widest possible range of types and meet performance expectations at the same time, and we relied on the compiler to select the best variant for the types involved.

In the asynchronous networking example, we removed many assumptions about the network stack from our Resource struct. Concrete resource values make no assumptions about the server's root domain or how to load data — they're just inert representations of API endpoints. Here, generic programming helps keep the resources simple and decoupled from the networking code. This also makes testing easier.

If you're interested in the theoretical details behind generic programming and how different languages facilitate it, we recommend Ronald Garcia et al.'s 2007 paper titled "An Extended Comparative Study of Language Support for Generic Programming."

Finally, generic programming in Swift wouldn't be possible without protocols. This brings us to our next chapter.

Protocols

In previous chapters, we saw how functions and generics can help us write very dynamic programs. Protocols work together with functions and generics to enable even more dynamic behavior.

Swift protocols aren't unlike Objective-C protocols. They can be used for delegation, and they allow you to specify abstract interfaces (such as `IteratorProtocol` or `Sequence`). Yet, at the same time, they're very different from Objective-C protocols. For example, we can make structs and enums conform to protocols. Additionally, protocols can have associated types. We can even add method implementations to protocols using protocol extensions. We'll look at all these things in the section on protocol-oriented programming.

Protocols allow for dynamic dispatch: the correct method implementation is selected at runtime based on the type of the receiver. However, when a method is or isn't dynamically dispatched is sometimes unintuitive, and this can lead to surprising behavior. We'll look at this issue in the next section.

Regular protocols can be used either as type constraints or as standalone types. Protocols with associated types and protocols with `Self` requirements, however, are special: we can't use them as standalone types, so something like `let x: Equatable` isn't allowed; we can only use them as type constraints, such as `func f<T: Equatable>(x: T)`. This may sound like a small limitation, but it makes protocols with associated types almost entirely different things in practice. We'll take a closer look at this later. We'll also discuss type erasers (such as AnyIterator) as a way of making it easier to work with protocols with associated types.

In object-oriented programming, subclassing is a powerful way to share code among multiple classes. A subclass inherits all the methods from its superclass and can choose to override some of the methods. For example, we could have an AbstractSequence class and subclasses such as Array and Dictionary. Doing so allows us to add methods to AbstractSequence, and all the subclasses would automatically inherit those methods.

Yet in Swift, the code sharing in Sequence is implemented using protocols and protocol extensions. In this way, the Sequence protocol and its extensions also work with value types, such as structs and enums, which don't support subclassing.

Not relying on subclassing also makes types more flexible. In Swift (as in most other object-oriented languages), a class can only have a single superclass. When we create a class, we have to choose the superclass well, because we can only pick one; we can't subclass both AbstractSequence, and, say, Stream. This can sometimes be a problem. There are examples in Cocoa — e.g. with NSMutableAttributedString, where the designers had to choose between NSAttributedString and NSMutableString as a superclass.

Some languages have multiple inheritance, the most common being C++. But this leads to something called the "diamond problem." For example, if multiple inheritance were possible, we could envision an NSMutableAttributedString that inherits from both NSMutableString and NSAttributedString. But what happens if both of those classes override a method of NSString? You could deal with it by just picking one of the methods. But what if that method is isEqual:? Providing good behavior for multiple inheritance is really hard.

Because multiple inheritance is so complicated, most languages don't support it. Yet many languages do support conforming to multiple protocols, which doesn't have the same problems. In Swift, we can conform to multiple protocols, and the compiler warns us when the use of a method is ambiguous.

Protocol extensions are a way of sharing code without sharing base classes. Protocols define a minimal viable set of methods for a type to implement. Extensions can then build on these minimal methods to implement more complex features.

For example, to implement a generic algorithm that sorts any sequence, you need two things. First, you need a way to iterate over the elements. Second, you need to be able to compare the elements. That's it. There are no demands as to how the elements are held. They could be in a linked list, an array, or any iterable container. What they are doesn't matter, either — they could be strings, integers, dates, or people. As long as you write down the two aforementioned constraints in the type system, you can implement sort:

```swift
extension Sequence where Element: Comparable {
    func sorted() -> [Element]
}
```

To implement an in-place sort, you need more building blocks. More specifically, you need index-based access to the elements rather than just linear iteration. Collection captures this and MutableCollection adds mutation to it. Finally, you need to compare and offset indices in constant time. RandomAccessCollection allows for that. This may sound complex, but it captures the prerequisites needed to perform an in-place sort:

```swift
extension MutableCollection where
    Self: RandomAccessCollection, Element: Comparable {
        mutating func sort()
}
```

Minimal capabilities described by protocols compose well. You can add different capabilities of different protocols to a type, bit by bit. We saw this in the collection protocols chapter when we first built a List type by giving it a single method, cons. Without changing the original List struct, we made it conform to Sequence. In fact, we could've done this even if we weren't the original authors of this type, using a technique called *retroactive modeling*. By adding conformance to Sequence, we get all the extension methods of Sequence for free.

Adding new shared features via a common superclass isn't this flexible. You can't just decide later to add a new common base class to many different classes; that would be a major refactoring. And if you aren't the owner of these subclasses, you can't do it at all!

Subclasses have to know which methods they can override without breaking the superclass. For example, when a method is overridden, a subclass might need to call the superclass method at the right moment: either at the beginning, somewhere in the middle, or at the end of the method. This moment is often unspecified. Also, by overriding the wrong method, a subclass might break the superclass without warning.

Protocol-Oriented Programming

In a graphical application, we might want different render targets: for example, we can either render graphics in a Core Graphics CGContext or create an SVG file. To start, we'll define a protocol that describes the minimum functionality of our drawing API:

```
protocol Drawing {
    mutating func addEllipse(rect: CGRect, fill: UIColor)
    mutating func addRectangle(rect: CGRect, fill: UIColor)
}
```

One of the most powerful features of protocols is that we can retroactively modify any type to add conformance to a protocol. For CGContext, we can add an extension that makes it conform to the Drawing protocol:

```
extension CGContext: Drawing {
    func addEllipse(rect: CGRect, fill: UIColor) {
        setFillColor(fill.cgColor)
        fillEllipse(in: rect)
    }

    func addRectangle(rect: CGRect, fill fillColor: UIColor) {
        setFillColor(fillColor.cgColor)
        fill(rect)
    }
}
```

To represent an SVG file, we create an SVG struct. It contains an XMLNode with children and has a single method, append, which appends a child node to the root node. (We've left out the definition of XMLNode here.)

```
struct SVG {
    var rootNode = XMLNode(tag: "svg")
    mutating func append(node: XMLNode) {
        rootNode.children.append(node)
    }
}
```

Rendering into an SVG means we have to append a node for each element. We use a few simple extensions: the svgAttributes property on CGRect creates a dictionary that matches the SVG specification. String.init(hexColor:) takes a UIColor and turns it into a hexadecimal string (such as "#010100"). With these helpers, adding Drawing conformance is straightforward:

```
extension SVG: Drawing {
    mutating func addEllipse(rect: CGRect, fill: UIColor) {
        var attributes: [String:String] = rect.svgAttributes
        attributes["fill"] = String(hexColor: fill)
        append(node: XMLNode(tag: "ellipse", attributes: attributes))
    }

    mutating func addRectangle(rect: CGRect, fill: UIColor) {
        var attributes: [String:String] = rect.svgAttributes
        attributes["fill"] = String(hexColor: fill)
        append(node: XMLNode(tag: "rect", attributes: attributes))
    }
}
```

We can now write drawing code that's independent of the rendering target; the following code only assumes that our context variable conforms to the Drawing protocol. If we initialized context with a CGContext instead, we wouldn't need to change any of the code:

```
var context: Drawing = SVG()
let rect1 = CGRect(x: 0, y: 0, width: 100, height: 100)
let rect2 = CGRect(x: 0, y: 0, width: 50, height: 50)
context.addRectangle(rect: rect1, fill: .yellow)
context.addEllipse(rect: rect2, fill: .blue)
context
/*
<svg>
<rect cy="0.0" fill="#010100" ry="100.0" rx="100.0" cx="0.0"/>
<ellipse cy="0.0" fill="#000001" ry="50.0" rx="50.0" cx="0.0"/>
</svg>
*/
```

Protocol Extensions

Another powerful feature of Swift protocols is the possibility to extend a protocol with full method implementations; you can do this to both your own protocols and existing protocols. For example, we could add a method to Drawing that, given a center point and a radius, renders a circle:

```
extension Drawing {
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {
        let diameter = radius * 2
        let origin = CGPoint(x: center.x - radius, y: center.y - radius)
        let size = CGSize(width: diameter, height: diameter)
        let rect = CGRect(origin: origin, size: size)
        addEllipse(rect: rect, fill: fill)
    }
}
```

By adding addCircle in an extension, we can use it both with CGContext and our SVG type.

Note that code sharing using protocols has several advantages over code sharing using inheritance:

- We're not forced to use a specific superclass.
- We can conform existing types to a protocol (e.g. we made CGContext conform to Drawing). Subclassing isn't as flexible; if CGContext were a class, we couldn't retroactively change its superclass.
- Protocols work with both structs and classes, but structs can't have superclasses.
- Finally, when dealing with protocols, we don't have to worry about overriding methods or calling super at the right moment.

Overriding Methods in Protocol Extensions

As the author of a protocol, you have two options when you add a protocol method in an extension. First, you can choose to only add it in an extension, as we did above with addCircle. Or, you could also add the method declaration to the protocol definition itself, making the method a *protocol requirement*. Protocol requirements are dispatched dynamically, whereas methods that are only defined in an extension use static dispatch. The difference is subtle but important, both as a protocol author and as a user of conforming types.

Let's look at an example. In the previous section, we added addCircle as an extension of the Drawing protocol, but we didn't make it a requirement. If we want to provide a more specific version of addCircle for the SVG type, we can just "override" the method:

```
extension SVG {
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {
        var attributes: [String:String] = [
          "cx": "\(center.x)",
          "cy": "\(center.y)",
          "r": "\(radius)",
        ]
        attributes["fill"] = String(hexColor: fill)
        append(node: XMLNode(tag: "circle", attributes: attributes))
    }
}
```

If we now create an instance of SVG and call addCircle on it, it behaves as you'd expect: the compiler will pick the most specific version of addCircle, which is the version that's defined in the extension on SVG. We can see that it correctly uses the circle tag:

```
var sample = SVG()
sample.addCircle(center: .zero, radius: 20, fill: .red)
print(sample)
/*
<svg>
<circle cy="0.0" fill="#010000" r="20.0" cx="0.0"/>
</svg>
*/
```

Now, just like above, we create another SVG instance; the only difference is that we explicitly cast the variable to the Drawing type. What'll happen if we call addCircle on this Drawing-that's-really-an-SVG? Most people would probably expect that this call would be dispatched to the same implementation on SVG, but that's not the case:

```
var otherSample: Drawing = SVG()
otherSample.addCircle(center: .zero, radius: 20, fill: .red)
print(otherSample)
/*
<svg>
<ellipse cy="-20.0" fill="#010000" ry="40.0" rx="40.0" cx="-20.0"/>
</svg>
*/
```

It returned an ellipse element, and not the circle we were expecting. It turns out it used the addCircle method from the protocol extension and not the method from the SVG extension. When we defined otherSample as a variable of type Drawing, the compiler automatically boxed the SVG value in a type that represents the protocol. This box is called an *existential container*, the details of which we'll look into later in this chapter. For now, let's consider the behavior: when we call addCircle on our existential container, the method is statically dispatched, i.e. it always uses the extension on Drawing. If it were dynamically dispatched, it would've taken the type of the receiver (SVG) into account.

To make addCircle dynamically dispatched, we add it as a protocol requirement:

```
protocol Drawing {
    mutating func addEllipse(rect: CGRect, fill: UIColor)
    mutating func addRectangle(rect: CGRect, fill: UIColor)
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor)
}
```

We can still provide a default implementation, just like before. And also like before, types are free to override addCircle. Because it's now part of the protocol definition, it'll be dynamically dispatched — at runtime, depending on the dynamic type of the receiver, the existential container will call the custom implementation if one exists. If it doesn't exist, it'll use the default implementation from the protocol extension. The addCircle method has become a *customization point* for the protocol.

The Swift standard library uses this technique a lot. A protocol like Sequence has dozens of requirements, yet almost all have default implementations. A conforming type can customize the default implementations because the methods are dynamically dispatched, but it doesn't have to.

Two Types of Protocols

As we stated in the introduction of this chapter, protocols with associated types are different from regular protocols. The same is true for protocols with a Self requirement, i.e. those that refer to Self anywhere in their definition. In Swift 4, these protocols can't be used as standalone types. This restriction will probably be lifted in a future version once the full generics system is implemented, but until then, we have to deal with the limitations.

One of the simplest examples of a protocol with an associated type is IteratorProtocol. It has a single associated type, Element, and a single function, next(), which returns an optional value of that type:

```
public protocol IteratorProtocol {
    associatedtype Element
    public mutating func next() -> Element?
}
```

In the chapter on collection protocols, we showed an example of a type that conforms to IteratorProtocol. This iterator simply returns 1 each time it's called:

```
struct ConstantIterator: IteratorProtocol {
    mutating func next() -> Int? {
        return 1
    }
}
```

As we've seen, IteratorProtocol forms the base for the collection protocols. Unlike IteratorProtocol, the Collection protocol doesn't have a simple definition:

```
protocol Collection: _Indexable, Sequence {
    associatedtype IndexDistance = Int
    associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>
    // ... Method definitions and more associated types
}
```

Let's look at some of the important parts of the definition above. The collection protocol inherits from _Indexable and Sequence. Because protocol inheritance doesn't have the same problems as inheritance through subclassing, we can compose multiple protocols:

```
protocol Collection: _Indexable, Sequence {
```

Next up, we have two associated types: IndexDistance and Iterator. Both have a default value: IndexDistance is just an Int, and Iterator is an IndexingIterator. Note that we can use Self for the generic type parameter of IndexingIterator. Both types also have constraints: IndexDistance needs to conform to the SignedInteger protocol (this is defined in _Indexable), and Iterator needs to conform to IteratorProtocol (this is defined in Sequence):

```
associatedtype IndexDistance: SignedInteger = Int
associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>
```

There are two options when we make our own types conform to the Collection protocol. We can either use the default associated types, or we can assign our own associated types (for example, in the collection protocols chapter, we made Words have a custom associated type for SubSequence). If we decide to stick with the default associated types, we get a lot of functionality for free. For example, there's a conditional protocol extension that adds an implementation of makeIterator() when the Iterator type isn't overridden:

```
extension Collection where Iterator == IndexingIterator<Self> {
    func makeIterator() -> IndexingIterator<Self>
}
```

There are many more conditional extensions, and you can also add your own. As we mentioned earlier, it can be challenging to see which methods you should implement in order to conform to a protocol. Because many protocols in the standard library have default values for the associated types and conditional extensions that match those associated types, you often only have to implement a handful of methods, even for protocols that have dozens of requirements. The (small) downside is that it can be difficult to determine what the minimal set of methods is that a conforming type has to implement to make the compiler happy. To address this, the standard library has documented it in a section, "Conforming to the Collection Protocol." If you write a custom protocol with more than a few methods, you should consider adding a similar section to your documentation.

Type Erasers

In the previous section, we were able to use the Drawing protocol as a type. However, with IteratorProtocol, this isn't (yet) possible, because it has an associated type. The compile error says:

“Protocol ‘IteratorProtocol’ can only be used as a generic constraint because it has Self or associated type requirements.”

```
let iterator: IteratorProtocol = ConstantIterator() // Error
```

In a way, IteratorProtocol is an incomplete type; we’d have to specify the associated type as well in order for this to be meaningful.

> *The Swift Core Team has stated that they want to support generalized existentials. This feature would allow for using protocols with associated types as standalone values, and it would also eliminate the need to write type erasers. For more information about what to expect in the future, see the Swift Generics Manifesto.*

In a future version of Swift, we might be able to solve this by saying something like the following:

```
let iterator: Any<IteratorProtocol where .Element == Int> = ConstantIterator()
```

Currently, we can’t yet express this. We can, however, use IteratorProtocol as a constraint for a generic parameter:

```
func nextInt<I: IteratorProtocol>(iterator: inout I) -> Int?
    where I.Element == Int {
        return iterator.next()
}
```

Similarly, we can store an iterator in a class or struct. The limitation is the same, in that we can only use it as a generic constraint, and not as a standalone type:

```
class IteratorStore<I: IteratorProtocol> where I.Element == Int {
    var iterator: I

    init(iterator: I) {
        self.iterator = iterator
    }
}
```

This works, but it has a drawback: the specific type of the stored iterator "leaks out" through the generic parameter. In the current type system, we can't express "any iterator, as long as the element type is Int." This is a problem if you want to, for example, put multiple IteratorStores into an array. All elements in an array must have the same type, and that includes any generic parameters; it's not possible to create an array that can store both IteratorStore<ConstantIterator> and IteratorStore<FibsIterator>.

Luckily, there are two ways around this — one is easy, the other one more efficient (but hacky). The process of removing a specific type (such as the iterator) is called *type erasure*.

In the easy solution, we implement a wrapper class. Instead of storing the iterator directly, the class stores the iterator's next function. To do this, we must first copy the iterator parameter to a local var variable so that we're allowed to call its next method (which is mutating). We then wrap the call to next() in a closure expression and assign that closure to a property. We used a class to signal that IntIterator has reference semantics:

```
class IntIterator {
    var nextImpl: () -> Int?

    init<I: IteratorProtocol>(_ iterator: I) where I.Element == Int {
        var iteratorCopy = iterator
        self.nextImpl = { iteratorCopy.next() }
    }
}
```

Now, in our IntIterator, the concrete type of the iterator (e.g. ConstantIterator) is only specified when creating an instance. After that, the concrete type is hidden, captured by the closure. We can create an IntIterator with any kind of iterator, as long as the elements are integers:

```
var iter = IntIterator(ConstantIterator())
iter = IntIterator([1,2,3].makeIterator())
```

The code above allows us to specify the associated type constraints (e.g. iter contains an iterator with Int elements) using Swift's current type system. Our IntIterator can also easily conform to the IteratorProtocol (and the inferred associated type is Int):

```
extension IntIterator: IteratorProtocol {
    func next() -> Int? {
        return nextImpl()
    }
}
```

In fact, by abstracting over Int and adding a generic parameter for the iterator's element type, we can change IntIterator to work just like the standard library's AnyIterator does:

```
class AnyIterator<A>: IteratorProtocol {
    var nextImpl: () -> A?

    init<I: IteratorProtocol>(_ iterator: I) where I.Element == A {
        var iteratorCopy = iterator
        self.nextImpl = { iteratorCopy.next() }
    }

    func next() -> A? {
        return nextImpl()
    }
}
```

The concrete iterator type (I) is only specified in the initializer, and after that, it's "erased."

From this refactoring, we can come up with a simple algorithm for creating a type eraser. First, we create a struct or class named AnyProtocolName. Then, for each associated type, we add a generic parameter. Next, for each of the protocol's methods, we store the implementation in a property on AnyProtocolName. Finally, we write an initializer that's generic over the concrete type we want to erase; its task is to capture the object that's passed in in closures that get assigned to the properties.

For a simple protocol like IteratorProtocol, this only takes a few lines of code, but for more complex protocols (such as Sequence), this is quite a lot of work. Even worse, the size of the object or struct will increase linearly with each protocol method (because a new property is added for each method).

The standard library takes a different approach to erasing types: it uses a class hierarchy to hide the concrete iterator type in a subclass, while the client-facing base class is only generic over the element type.

To see how this works, we start by creating a simple abstract class that conforms to IteratorProtocol. Its generic type is the Element of the iterator, and the implementation will simply crash:

```swift
class IteratorBox<Element>: IteratorProtocol {
    func next() -> Element? {
        fatalError("This method is abstract.")
    }
}
```

Then, we create another class, IteratorBoxHelper, which is also generic. Here, the generic parameter is the specific iterator type (for example, ConstantIterator). The purpose of this class is to store the underlying iterator in a property. The next method simply forwards to the iterator's next method:

```swift
class IteratorBoxHelper<I: IteratorProtocol> {
    var iterator: I
    init(iterator: I) {
        self.iterator = iterator
    }

    func next() -> I.Element? {
        return iterator.next()
    }
}
```

Now for the hacky part. We change IteratorBoxHelper so that it's a subclass of IteratorBox, and the two generic parameters are constrained in such a way that IteratorBox gets I's element as the generic parameter:

```
class IteratorBoxHelper<I: IteratorProtocol>: IteratorBox<I.Element> {
    var iterator: I
    init(_ iterator: I) {
        self.iterator = iterator
    }

    override func next() -> I.Element? {
        return iterator.next()
    }
}
```

The "magic" happens in the initializer of IteratorBoxHelper. IteratorBox can't have a property to store the wrapped iterator directly, because then it would need to be generic over the concrete iterator type, which is exactly what we want to avoid. The solution is to hide this property (and with it, its concrete type) in the subclass, which can be generic over the concrete iterator type. IteratorBox is then able to be generic only over the element type.

This allows us to create a value of IteratorBoxHelper and use it as an IteratorBox, effectively erasing the type of I:

```
let iter: IteratorBox<Int> = IteratorBoxHelper(ConstantIterator())
```

In the standard library, the IteratorBox and IteratorBoxHelper are then made private, and yet another wrapper (AnyIterator) makes sure that these implementation details are hidden from the public interface.

Protocols with Self Requirements

Protocols with Self requirements behave in a way similar to protocols with associated types. One of the simplest protocols with a Self requirement is Equatable. It has a single method (in the form of an operator) that compares two elements:

```
protocol Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool
}
```

Implementing Equatable for your own type isn't too hard. For example, if we have a simple MonetaryAmount struct, we can compare two values by comparing their properties:

```
struct MonetaryAmount: Equatable {
    var currency: String
    var amountInCents: Int

    static func ==(lhs: MonetaryAmount, rhs: MonetaryAmount) -> Bool {
        return lhs.currency == rhs.currency &&
                lhs.amountInCents == rhs.amountInCents
    }
}
```

We can't simply declare a variable with Equatable as its type:

```
// Error: 'Equatable' can only be used as a generic constraint
// because it has Self or associated type requirements
let x: Equatable = MonetaryAmount(currency: "EUR", amountInCents: 100)
```

This suffers from the same problems as associated types: from this (incorrect) declaration, it's unclear what the Self type should be. For example, if it were possible to use Equatable as a standalone type, you could also write this:

```
let x: Equatable = MonetaryAmount(currency: "EUR", amountInCents: 100)
let y: Equatable = "hello"
x == y
```

There's no definition of == that takes a monetary amount and a string. How would you compare the two? However, we can use Equatable as a generic constraint. For example, we could write a free function, allEqual, that checks whether all elements in an array are equal:

```
func allEqual<E: Equatable>(x: [E]) -> Bool {
    guard let firstElement = x.first else { return true }
    for element in x {
        guard element == firstElement else { return false }
    }
    return true
}
```

Or we could use it as a constraint when writing an extension on Collection:

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        guard let firstElement = first else { return true }
        for element in self {
            guard element == firstElement else { return false }
        }
        return true
    }
}
```

The == operator is defined as a static function of the type. In other words, it's not a member function, and it's statically dispatched. Unlike member functions, we can't override it. If you have a class that implements Equatable (for example, NSObject), you might get unexpected behavior when you create a subclass. For example, consider the following class:

```
class IntegerRef: NSObject {
    let int: Int
    init(_ int: Int) {
        self.int = int
    }
}
```

We can define a version of == that compares two IntegerRefs by comparing their int properties:

```
func ==(lhs: IntegerRef, rhs: IntegerRef) -> Bool {
    return lhs.int == rhs.int
}
```

If we create two IntegerRef objects, we can compare them, and everything works as expected:

```
let one = IntegerRef(1)
let otherOne = IntegerRef(1)
one == otherOne // true
```

However, if we use them as NSObjects, the == of NSObject is used (which uses === under the hood to check if the references point to the same object). Unless you're aware of the static dispatch behavior, the result might come as a surprise:

```
let two: NSObject = IntegerRef(2)
let otherTwo: NSObject = IntegerRef(2)
two == otherTwo // false
```

Protocol Internals

As we mentioned earlier, when we create a variable that has a protocol type, it's wrapped in a box called an existential container. Let's take a closer look at this.

Consider the following pair of functions. Both take a value that conforms to CustomStringConvertible and return the size of the value's type. The only difference is that one function uses the protocol as a generic constraint, whereas the other uses it as a type:

```
func f<C: CustomStringConvertible>(_ x: C) -> Int {
    return MemoryLayout.size(ofValue: x)
}
func g(_ x: CustomStringConvertible) -> Int {
    return MemoryLayout.size(ofValue: x)
}
```

```
f(5) // 8
g(5) // 40
```

Where does the size difference of 8 bytes versus 40 bytes come from? Because f takes a generic parameter, the integer 5 is passed straight to the function without any kind of boxing. This is reflected in its size of 8 bytes, which is the size of Int on a 64-bit system. For g, the integer is wrapped in an existential container. For regular (i.e. non-class-constrained) protocols, an *opaque existential container* is used. Opaque existential containers contain a buffer (the size of three pointers, or 24 bytes) for the value; some metadata (one pointer, 8 bytes); and a number of *witness tables* (zero or more pointers, 8 bytes each). If the value doesn't fit into the buffer, it's stored on the heap, and the buffer contains a reference to the storage location. The metadata contains information about the type (so that it can be used with a conditional cast, for example).

The witness table is what makes dynamic dispatch possible. It encodes the implementation of a protocol for a specific type: for each method in the protocol, the table contains an entry that points to the implementation for that specific type. Sometimes this is also called a *vtable*. In a way, when we created the first version of AnyIterator, we manually wrote a witness table.

Given the witness table, the surprising behavior of addCircle at the beginning of this chapter makes a lot more sense. Because addCircle wasn't part of the protocol definition (i.e. it wasn't a *requirement*), it wasn't in the witness table either. Therefore, the compiler had no choice but to statically call the protocol's default implementation. Once we made addCircle a protocol requirement, it was also added to the witness table, and therefore called using dynamic dispatch.

The size of the opaque existential container depends on the number of witness tables: it contains one witness table per protocol. For example, Any is a typealias for an empty protocol that has no witness tables at all:

```
typealias Any = protocol<>

MemoryLayout<Any>.size // 32
```

If we combine multiple protocols, one 8-byte chunk gets added per protocol. So combining four protocols adds 32 bytes:

```
protocol Prot { }
protocol Prot2 { }
protocol Prot3 { }
protocol Prot4 { }
typealias P = Prot & Prot2 & Prot3 & Prot4
MemoryLayout<P>.size // 64
```

For class-only protocols, there's a special existential container called a *class existential container*, which only has the size of two words (plus one word per additional witness table) — one for the metadata and one (instead of three) for a reference to the class:

```
protocol ClassOnly: AnyObject {}
MemoryLayout<ClassOnly>.size // 16
```

Objective-C protocols that get imported into Swift don't have additional metadata. Therefore, variables whose type is an Objective-C protocol don't have to be wrapped in an existential container; they only consist of a plain pointer to their class:

```
MemoryLayout<NSObjectProtocol>.size // 8
```

Performance Implications

Existential containers add a level of indirection and therefore generally cause a performance decrease over generic parameters (assuming the compiler can specialize the generic code). In addition to the possibly slower method dispatch, the existential container also acts as a barrier that prevents many compiler optimizations. For the most part, worrying about this overhead is premature optimization. However, if you need maximum performance in a tight loop, it'll be more efficient to use a generic parameter rather than a parameter that has a protocol type. This way, you can avoid the implicit protocol box.

If you try to pass [String] (or any other type) into a function that takes [Any] (or any other array of a protocol rather than a specific type), the compiler will emit code that maps over the entire array and boxes every value, effectively making the function call itself — not the function body — an $O(n)$ operation (where n is the number of elements in the array). Again, this won't be an issue in most cases, but if you need to write highly performant code, you might choose to write your function with a generic parameter rather than a protocol type:

```
// Implicit boxing
func printProtocol(array: [CustomStringConvertible]) {
    print(array)
}

// No boxing
func printGeneric<A: CustomStringConvertible>(array: [A]) {
    print(array)
}
```

Recap

Protocols in Swift are important building blocks. They allow us to write flexible code that decouples the interface and implementation. We looked at the two different kinds of protocols and how they're implemented. We used type erasers and looked at the performance differences between protocol types and generic types with a protocol constraint. We also looked at unexpected behavior due to the differences between static and dynamic dispatch.

Protocols had a big impact on the Swift community. Yet we also want to warn against overusing them. Sometimes, using a simple concrete type like a struct or a class is much easier to understand than a protocol, and this increases the readability of your code. However, there are also times when using a protocol can increase readability of your code — for example, when you deal with legacy APIs retrofitted to a protocol.

One of the big upsides of using protocols is that they provide a *minimal viable interface*; a well-designed protocol specifies the minimum set of requirements conforming types must meet to be able to work with algorithms that were designed for the protocol. Also, protocols can make it easier to write test code. Rather than having to set up a complicated tree of dependencies, we can just create a simple test type that conforms to the protocol.

Interoperability

One of Swift's greatest strengths is the low friction when interoperating with C and Objective-C. Swift can automatically bridge Objective-C types to native Swift types, and it can even bridge with many C types. This allows us to use existing libraries and provide a nice interface on top.

In this chapter, we'll create a wrapper around the CommonMark library. CommonMark is a formal specification for Markdown, a popular syntax for formatting plain text. If you've ever written a post on GitHub or Stack Overflow, you've probably used Markdown. After this practical example, we'll take a look at the tools the standard library provides for working with memory, and we'll see how they can be used to interact with C code.

Hands-On: Wrapping CommonMark

Swift's ability to call into C code allows us to take advantage of the abundance of existing C libraries. Writing a wrapper around an existing library's interface in Swift is often much easier and involves less work than building something from scratch; meanwhile, users of our wrapper will see no difference in terms of type safety or ease of use compared to a fully native solution. All we need to start is the dynamic library and its C header files.

Our example, the CommonMark C library, is a reference implementation of the CommonMark spec that's both fast and well tested. We'll take a layered approach in order to make CommonMark accessible from Swift. First, we'll create a very thin

Swift class around the opaque types the library exposes. Then, we'll wrap this class with Swift enums in order to provide a more idiomatic API.

Setup

Setting up a C library so that it can be imported from a Swift project is a little complicated. Here's a quick rundown of the required steps.

The first step is to install the cmark library. We use Homebrew as our package manager on macOS to do this:

```
$ brew install cmark
```

At the time of writing, cmark 0.28.2 was the most recent version.

In C, you'd now `#include` one or more of the library's header files to make their declarations visible to your own code. Swift can't handle C header files directly; it expects dependencies to be *modules*. For a C or Objective-C library to be visible to the Swift compiler, the library must provide a *module map* in the Clang modules format. Among other things, the module map lists the header files that make up the module.

Since cmark doesn't come with a module map, your next task is to make one. You'll create a package for the Swift Package Manager that won't contain any code; its only purpose is to act as the module wrapper for the cmark library.

Create a directory for the package and then call `swift package init`, telling the package manager to create the scaffolding for a *system module*:

```
$ mkdir Ccmark
$ cd Ccmark
$ swift package init --type system-module
```

In SwiftPM lingo, *system packages* are libraries installed by systemwide package managers, such as Homebrew, or APT on Linux. A system module is any SwiftPM package that refers to such a library. By convention, the names of pure wrapper modules such as this should be prefixed with C.

Edit Package.swift to look like this:

```
// swift-tools-version:4.0
import PackageDescription

let package = Package(
    name: "Ccmark",
    pkgConfig: "libcmark",
    providers: [
        .brew(["cmark"])
    ]
)
```

The pkgConfig parameter specifies the name of the config file where the package manager can find the header and library search paths for the imported library. The providers directive is optional. It's an installation hint the package manager can display when the target library isn't installed.

Before you edit the module map, create a C header file named shim.h. It should contain only the following line:

```
#include <cmark.h>
```

Finally, the module.modulemap file should look like this:

```
module Ccmark [system] {
    header "shim.h"
    link "cmark"
    export *
}
```

The shim header works around the limitation that module maps must contain absolute paths. Alternatively, you could've omitted the shim and specified the cmark header directly in the module map, as in `header "/usr/local/include/cmark.h"`. But then the path of `cmark.h` would be hardcoded into the module map. With the shim, the package manager reads the correct header search path from the pkg-config file and adds it to the compiler invocation.

The final step for the Ccmark package is to commit everything to a Git repository:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

This is necessary because you'll now create a second package which imports `Ccmark`, and the package manager requires a Git branch or tag name for each dependency.

Create another directory next to the `Ccmark` directory and call `swift package init` once more, this time for an executable:

```
$ cd ..
$ mkdir CommonMarkExample
$ cd CommonMarkExample
$ swift package init --type executable
```

You'll need to add the `Ccmark` dependency to the package manifest:

```
// swift-tools-version:4.0
import PackageDescription

let package = Package(
    name: "CommonMarkExample",
    dependencies: [
        .package(url: "../Ccmark", .branch("master")),
    ],
    targets: [
        .target(
            name: "CommonMarkExample",
            dependencies: []),
    ]
)
```

Notice that we're using a relative file system path to refer to Ccmark. If you want to make this accessible to other team members, push the Ccmark repository to a server and replace its URL here.

Now you should be able to import Ccmark and call any cmark API. Add the following snippet to main.swift for a quick test to see if everything works:

```
import Ccmark

let markdown = "*Hello World*"
let cString = cmark_markdown_to_html(markdown, markdown.utf8.count, 0)!
let html = String(cString: cString)
print(html)
```

Back in Terminal, run the program:

```
$ swift run
```

If you see <p><em>Hello World</em></p> as the output, you just successfully called a C function from Swift! Now that we have a working installation, we can start writing our Swift wrapper.

Wrapping the C Library

Let's begin by wrapping a single function with a nicer interface. The cmark_markdown_to_html function takes in Markdown-formatted text and returns the resulting HTML code in a string. The C interface looks like this:

```
/// Convert 'text' (assumed to be a UTF-8 encoded string with length
/// 'len') from CommonMark Markdown to HTML, returning a null-terminated,
/// UTF-8-encoded string. It is the caller's responsibility
/// to free the returned buffer.
char *cmark_markdown_to_html(const char *text, size_t len, int options);
```

When Swift imports this declaration, it presents the C string in the first parameter as an `UnsafePointer` to a number of `Int8` values. From the documentation, we know that these are expected to be UTF-8 code units. The `len` parameter takes the length of the string:

```
// The function's interface in Swift
func cmark_markdown_to_html
    (_ text: UnsafePointer<Int8>!, _ len: Int, _ options: Int32)
    -> UnsafeMutablePointer<Int8>!
```

We want our wrapper function to work with Swift strings, of course, so you might think that we need to convert the Swift string into an `Int8` pointer before passing it to `cmark_markdown_to_html`. However, bridging between native and C strings is such a common operation that Swift will do this automatically. We do have to be careful with the `len` parameter, as the function expects the length of the UTF-8-encoded string in bytes, and not the number of characters. We get the correct value from the string's `utf8` view, and we can just pass in zero for the options:

```
func markdownToHtml(input: String) -> String {
    let outString = cmark_markdown_to_html(input, input.utf8.count, 0)!
    defer { free(outString) }
    return String(cString: outString)
}
```

Notice that we force-unwrap the string pointer the function returns. We can safely do this because we know that `cmark_markdown_to_html` always returns a valid string. By force-unwrapping inside the method, the library user can call the `markdownToHtml` method without having to worry about optionals — the result would never be `nil` anyway. This is something the compiler can't do automatically for us — C and Objective-C pointers without nullability annotations are always imported into Swift as implicitly unwrapped optionals.

> *The automatic bridging of native Swift strings to C strings assumes that the C function you want to call expects the string to be UTF-8 encoded. This is the correct choice in most cases, but if the C API assumes a different encoding, you can't use the automatic bridging. However, it's often quite easy to construct alternative formats. For example, if the C API expects an array of UTF-16 code points, you can use* `Array(string.utf16)`*. The Swift compiler will automatically bridge the Swift array to the expected C array, provided that the element types match.*

Wrapping the cmark_node Type

In addition to the straight HTML output, the `cmark` library also provides a way to parse a Markdown text into a structured tree of elements. For example, a simple text could be transformed into a list of block-level nodes such as paragraphs, quotes, lists, code blocks, headers, and so on. Some block-level elements contain other block-level elements (for example, quotes can contain multiple paragraphs), whereas others contain only inline elements (for example, a header can contain a part that's emphasized). No element can contain both (for example, the inline elements of a list item are always wrapped in a paragraph element).

The C library uses a single data type, `cmark_node`, to represent the nodes. It's opaque, meaning the authors of the library chose to hide its definition. All we see in the headers are functions that operate on or return pointers to `cmark_node`. Swift imports these pointers as `OpaquePointer`s. (We'll take a closer look at the differences between the many pointer types in the standard library, such as OpaquePointer and UnsafeMutablePointer, later in this chapter.)

> *In the future, we might be able to use the "import as member" feature of Swift to import these functions as methods on* cmark_node*. Apple has used this to provide more "object-oriented" APIs for Core Graphics and Grand Central Dispatch in Swift. It works by annotating the C source code with the* swift_name *attribute. Alas, this feature doesn't really work for the cmark library yet. It might work for your own C library. There are some bugs around import as member, for example, it doesn't always work with opaque pointers (which is exactly what's used in cmark).*

Let's wrap a node in a native Swift type to make it easier to work with. As we saw in the chapter on structs and classes, we need to think about value semantics whenever we create a custom type: is the type a value, or does it make sense for instances to have identity? In the former case, we should favor a struct or enum, whereas the latter requires a class. Our case is interesting: on one hand, the node of a Markdown document is a value — two nodes that have the same element type and contents should be indistinguishable, hence they shouldn't have identity. On the other hand, since we don't know the internals of cmark_node, there's no straightforward way to make a copy of a node, so we can't guarantee value semantics. For this reason, we start with a class. Later on, we'll write another layer on top of this class to provide an interface with value semantics.

Our class simply stores the opaque pointer and frees the memory cmark_node uses on deinit when there are no references left to an instance of this class. We only free memory at the document level, because otherwise we might free nodes that are still in use. Freeing the document will also automatically free all the children recursively. Wrapping the opaque pointer in this way will give us automatic reference counting for free:

```
public class Node {
    let node: OpaquePointer

    init(node: OpaquePointer) {
        self.node = node
    }

    deinit {
        guard type == CMARK_NODE_DOCUMENT else { return }
        cmark_node_free(node)
    }
}
```

The next step is to wrap the cmark_parse_document function, which parses a Markdown text and returns the document's root node. It takes the same arguments as cmark_markdown_to_html: the string, its length, and an integer describing parse options. The return type of the cmark_parse_document function in Swift is OpaquePointer, which represents the node:

```
func cmark_parse_document
    (_ buffer: UnsafePointer<Int8>!, _ len: Int, _ options: Int32)
    -> OpaquePointer!
```

We turn the function into an initializer for our class. Note that the function can return nil if parsing fails. Therefore, our initializer should be failable and return nil (the optional value, not the null pointer) if this occurs:

```
public init?(markdown: String) {
    guard let node = cmark_parse_document(markdown,
        markdown.utf8.count, 0) else { return nil }
    self.node = node
}
```

As mentioned above, there are a couple of interesting functions that operate on nodes. For example, there's one that returns the type of a node, such as paragraph or header:

```
cmark_node_type cmark_node_get_type(cmark_node *node);
```

In Swift, it looks like this:

```
func cmark_node_get_type(_ node: OpaquePointer!) -> cmark_node_type
```

cmark_node_type is a C enum that has cases for the various block-level and inline elements that are defined in Markdown, as well as one case to signify errors:

```
typedef enum {
    // Error status
    CMARK_NODE_NONE,

    // Block
    CMARK_NODE_DOCUMENT,
    CMARK_NODE_BLOCK_QUOTE,
    ...

    // Inline
    CMARK_NODE_TEXT,
    CMARK_NODE_EMPH,
    ...
} cmark_node_type;
```

Swift imports plain C enums as structs containing a single UInt32 property. Additionally, for every case in an enum, a global variable is generated:

```
struct cmark_node_type : RawRepresentable, Equatable {
    public init(_ rawValue: UInt32)
    public init(rawValue: UInt32)
    public var rawValue: UInt32
}

var CMARK_NODE_NONE: cmark_node_type { get }
var CMARK_NODE_DOCUMENT: cmark_node_type { get }
...
```

Only enums marked with the NS_ENUM macro, used by Apple in its Objective-C frameworks, are imported as native Swift enumerations. Additionally, library authors can annotate their enum cases with swift_name to make them member variables.

In Swift, the type of a node should be a property of the Node data type, so we turn the cmark_node_get_type function into a computed property of our class:

```
var type: cmark_node_type {
    return cmark_node_get_type(node)
}
```

Now we can just write node.type to get an element's type.

There are a couple more node properties we can access. For example, if a node is a list, it can have one of two list types: bulleted or ordered. All other nodes have the list type "no list." Again, Swift represents the corresponding C enum as a struct, with a top-level variable for each case, and we can write a similar wrapper property. In this case, we also provide a setter, which will come in handy later in this chapter:

```
var listType: cmark_list_type {
    get { return cmark_node_get_list_type(node) }
    set { cmark_node_set_list_type(node, newValue) }
}
```

There are similar functions for all the other node properties (such as header level, fenced code block info, and link URLs and titles). These properties often only make sense for specific types of nodes, and we can choose to provide an interface either with an optional (e.g. for the link URL) or with a default value (e.g. the default header level is zero). This illustrates a major weakness of the library's C API that we can model much better in Swift. We'll talk more about this below.

Some nodes can also have children. For iterating over them, the CommonMark library provides the functions cmark_node_first_child and cmark_node_next. We want our Node class to provide an array of its children. To generate this array, we start with the first child and keep adding children until either cmark_node_first_child or cmark_node_next returns nil, signaling the end of the list. Note that this C null pointer automatically gets converted to an optional:

```
var children: [Node] {
    var result: [Node] = []
    var child = cmark_node_first_child(node)
    while let unwrapped = child {
        result.append(Node(node: unwrapped))
        child = cmark_node_next(child)
    }
    return result
}
```

We could also have chosen to return a lazy sequence rather than an array (for example, by using `sequence` or `AnySequence`). However, there's a problem with this: the node structure might change between creation and consumption of the sequence. In that case, the iterator for finding the next node would return wrong values, or even worse, crash. Depending on your use case, returning a lazily constructed sequence might be exactly what you want, but if your data structure can change, returning an array is a much safer choice.

With this simple wrapper class for nodes, accessing the abstract syntax tree produced by the CommonMark library from Swift becomes a lot easier. Instead of having to call functions like `cmark_node_get_list_type`, we can just write `node.listType` and get autocompletion and type safety. However, we aren't done yet. Even though the `Node` class feels much more native than the C functions, Swift allows us to express a node in an even more natural and safer way, using enums with associated values.

A Safer Interface

As we mentioned above, there are many node properties that only apply in certain contexts. For example, it doesn't make any sense to access the headerLevel of a list or the listType of a code block. Enumerations with associated values allow us to specify only the metadata that makes sense for each specific case. We'll create one enum for all possible inline elements, and another one for block-level items. That way, we can enforce the structure of a CommonMark document. For example, a plain text element just stores a String, whereas emphasis nodes contain an array of other inline elements. These enumerations will be the public interface to our library, turning the Node class into an internal implementation detail:

```swift
public enum Inline {
    case text(text: String)
    case softBreak
    case lineBreak
    case code(text: String)
    case html(text: String)
    case emphasis(children: [Inline])
    case strong(children: [Inline])
    case custom(literal: String)
    case link(children: [Inline], title: String?, url: String?)
    case image(children: [Inline], title: String?, url: String?)
}
```

Similarly, paragraphs and headers can only contain inline elements, whereas block quotations always contain other block-level elements. A list is defined as an array of list items, where each list item is represented by an array of Block elements:

```swift
public enum Block {
    case list(items: [[Block]], type: ListType)
    case blockQuote(items: [Block])
    case codeBlock(text: String, language: String?)
    case html(text: String)
    case paragraph(text: [Inline])
    case heading(text: [Inline], level: Int)
    case custom(literal: String)
    case thematicBreak
}
```

The ListType is just a simple enum that states whether a list is ordered or unordered:

```
public enum ListType {
    case unordered
    case ordered
}
```

Since enums are value types, this also lets us treat nodes as values by converting them to their enum representations. We follow the API Design Guidelines by using initializers for type conversions. We write two pairs of initializers: one pair creates Block and Inline values from the Node type, and another pair reconstructs a Node from these enums. This allows us to write functions that create or manipulate Inline or Block values and then later reconstruct a CommonMark document and use the C library to render it into HTML or back into Markdown text.

Let's start by writing an initializer that converts a Node into an Inline element. We switch on the node's type and construct the corresponding Inline value. For example, for a text node, we take the node's string contents, which we access through the literal property in the cmark library. We can safely force-unwrap literal because we know that text nodes always have this value, whereas other node types might return nil from literal. For example, emphasis and strong nodes only have child nodes and no literal value. To parse the latter, we map over the node's children and call our initializer recursively. Instead of duplicating that code, we create an inline function, inlineChildren, that only gets called when needed. The default case should never get reached, so we choose to trap the program if it does. This follows the convention that returning an optional or using throws should generally only be used for expected errors, and not to signify programmer errors:

```
extension Inline {
    init(_ node: Node) {
        let inlineChildren = { node.children.map(Inline.init) }
        switch node.type {
        case CMARK_NODE_TEXT:
            self = .text(text: node.literal!)
        case CMARK_NODE_SOFTBREAK:
            self = .softBreak
        case CMARK_NODE_LINEBREAK:
            self = .lineBreak
        case CMARK_NODE_CODE:
            self = .code(text: node.literal!)
        case CMARK_NODE_HTML_INLINE:
            self = .html(text: node.literal!)
        case CMARK_NODE_CUSTOM_INLINE:
            self = .custom(literal: node.literal!)
        case CMARK_NODE_EMPH:
            self = .emphasis(children: inlineChildren())
        case CMARK_NODE_STRONG:
            self = .strong(children: inlineChildren())
        case CMARK_NODE_LINK:
            self = .link(children: inlineChildren(), title: node.title, url: node.urlString)
        case CMARK_NODE_IMAGE:
            self = .image(children: inlineChildren(), title: node.title, url: node.urlString)
        default:
            fatalError("Unrecognized node:\(node.typeString)")
        }
    }
}
```

Converting block-level elements follows the same pattern. Note that block-level elements can have either inline elements, list items, or other block-level elements as children, depending on the node type. In the cmark_node syntax tree, list items get wrapped with an extra node. In the listItem property on Node, we remove that layer and directly return an array of block-level elements:

```
extension Block {
    init(_ node: Node) {
        let parseInlineChildren = { node.children.map(Inline.init) }
        let parseBlockChildren = { node.children.map(Block.init) }
        switch node.type {
        case CMARK_NODE_PARAGRAPH:
            self = .paragraph(text: parseInlineChildren())
        case CMARK_NODE_BLOCK_QUOTE:
            self = .blockQuote(items: parseBlockChildren())
        case CMARK_NODE_LIST:
            let type = node.listType == CMARK_BULLET_LIST ?
                ListType.unordered : ListType.ordered
            self = .list(items: node.children.map { $0.listItem }, type: type)
        case CMARK_NODE_CODE_BLOCK:
            self = .codeBlock(text: node.literal!, language: node.fenceInfo)
        case CMARK_NODE_HTML_BLOCK:
            self = .html(text: node.literal!)
        case CMARK_NODE_CUSTOM_BLOCK:
            self = .custom(literal: node.literal!)
        case CMARK_NODE_HEADING:
            self = .heading(text: parseInlineChildren(), level: node.headerLevel)
        case CMARK_NODE_THEMATIC_BREAK:
            self = .thematicBreak
        default:
            fatalError("Unrecognized node:\(node.typeString)")
        }
    }
}
```

Now, given a document-level Node, we can easily convert it into an array of Block elements. The Block elements are values: we can freely copy or change them without having to worry about references. This is very powerful for manipulating nodes. Since values, by definition, don't care how they were created, we can also create a Markdown syntax tree in code, from scratch, without using the CommonMark library at all. The types are much clearer too; you can't accidentally do things that wouldn't make sense — such as accessing the title of a list — as the compiler won't allow it. Aside from making your code safer, this is a very robust form of documentation — by just looking at the types, it's obvious how a CommonMark document is structured. And unlike comments, the compiler will make sure that this form of documentation is never outdated.

It's now very easy to write functions that operate on our new data types. For example, if we want to build a list of all the level one and two headers from a Markdown document for a table of contents, we can just loop over all children and check whether they are headers and have the correct level:

```
func tableOfContents(document: String) -> [Block] {
    let blocks = Node(markdown: document)?.children.map(Block.init) ?? []
    return blocks.filter {
        switch $0 {
        case .heading(_, let level) where level < 3: return true
        default: return false
        }
    }
}
```

Before we build more operations like this, let's tackle the inverse transformation: converting a Block back into a Node. We need this because we ultimately want to use the CommonMark library to generate HTML or other text formats from the Markdown syntax tree we've built or manipulated, and the library can only deal with cmark_node_type.

Our plan is to add two initializers on Node: one that converts an Inline value to a node, and another that handles Block elements. We start by extending Node with a new initializer that creates a new cmark_node from scratch with the specified type and children. Recall that we wrote a deinit, which frees the root node of the tree (and recursively, all its children). This deinit will make sure that the node we allocate here gets freed eventually:

```
extension Node {
    convenience init(type: cmark_node_type, children: [Node] = []) {
        self.init(node: cmark_node_new(type))
        for child in children {
            cmark_node_append_child(node, child.node)
        }
    }
}
```

We'll frequently need to create text-only nodes, or nodes with a number of children, so let's add three convenience initializers to make that easier:

```
extension Node {
    convenience init(type: cmark_node_type, literal: String) {
        self.init(type: type)
        self.literal = literal
    }
    convenience init(type: cmark_node_type, blocks: [Block]) {
        self.init(type: type, children: blocks.map(Node.init))
    }
    convenience init(type: cmark_node_type, elements: [Inline]) {
        self.init(type: type, children: elements.map(Node.init))
    }
}
```

Now we're ready to write the two conversion initializers. Using the initializers we just defined, it becomes very straightforward: we switch on the element and create a node with the correct type. Here's the version for inline elements:

```
extension Node {
    convenience init(element: Inline) {
        switch element {
        case .text(let text):
            self.init(type: CMARK_NODE_TEXT, literal: text)
        case .emphasis(let children):
            self.init(type: CMARK_NODE_EMPH, elements: children)
        case .code(let text):
            self.init(type: CMARK_NODE_CODE, literal: text)
        case .strong(let children):
            self.init(type: CMARK_NODE_STRONG, elements: children)
        case .html(let text):
            self.init(type: CMARK_NODE_HTML_INLINE, literal: text)
        case .custom(let literal):
            self.init(type: CMARK_NODE_CUSTOM_INLINE, literal: literal)
        case let .link(children, title, url):
            self.init(type: CMARK_NODE_LINK, elements: children)
            self.title = title
            self.urlString = url
        case let .image(children, title, url):
            self.init(type: CMARK_NODE_IMAGE, elements: children)
            self.title = title
            urlString = url
        case .softBreak:
            self.init(type: CMARK_NODE_SOFTBREAK)
        case .lineBreak:
            self.init(type: CMARK_NODE_LINEBREAK)
        }
    }
}
```

Creating a node from a block-level element is very similar. The only slightly more complicated case is lists. Recall that in the above conversion from Node to Block, we removed the extra node the CommonMark library uses to represent lists, so we need to add that back in here:

```
extension Node {
    convenience init(block: Block) {
        switch block {
        case .paragraph(let children):
            self.init(type: CMARK_NODE_PARAGRAPH, elements: children)
        case let .list(items, type):
            let listItems = items.map { Node(type: CMARK_NODE_ITEM, blocks: $0) }
            self.init(type: CMARK_NODE_LIST, children: listItems)
            listType = type == .unordered
                ? CMARK_BULLET_LIST
                : CMARK_ORDERED_LIST
        case .blockQuote(let items):
            self.init(type: CMARK_NODE_BLOCK_QUOTE, blocks: items)
        case let .codeBlock(text, language):
            self.init(type: CMARK_NODE_CODE_BLOCK, literal: text)
            fenceInfo = language
        case .html(let text):
            self.init(type: CMARK_NODE_HTML_BLOCK, literal: text)
        case .custom(let literal):
            self.init(type: CMARK_NODE_CUSTOM_BLOCK, literal: literal)
        case let .heading(text, level):
            self.init(type: CMARK_NODE_HEADING, elements: text)
            headerLevel = level
        case .thematicBreak:
            self.init(type: CMARK_NODE_THEMATIC_BREAK)
        }
    }
}
```

Finally, to provide a nice interface for the user, we define a public initializer that takes an array of block-level elements and produces a document node, which we can then render into one of the different output formats:

```
extension Node {
    public convenience init(blocks: [Block]) {
        self.init(type: CMARK_NODE_DOCUMENT, blocks: blocks)
    }
}
```

Now we can go in both directions: we can load a document, convert it into `[Block]` elements, modify those elements, and turn them back into a `Node`. This allows us to write programs that extract information from Markdown or even change the Markdown dynamically. By first creating a thin wrapper around the C library (the `Node` class), we abstracted the conversion from the underlying C API. This allowed us to focus on providing an interface that feels like idiomatic Swift. The entire project is available on GitHub.

An Overview of Low-Level Types

There are many types in the standard library that provide low-level access to memory. Their sheer number can be overwhelming, as can daunting names like `UnsafeMutableRawBufferPointer`. The good news is that they're named consistently, so each type's purpose can be deduced from its name. Here are the most important naming parts:

- A **managed** type has automatic memory management. The compiler will allocate, initialize, and free the memory for you.

- An **unsafe** type doesn't provide automated memory management (as opposed to *managed*) . You have to allocate, initialize, deallocate, and deinitialize the memory explicitly.

- A **buffer** type works on multiple (contiguously stored) elements rather than a single element and provides a

Collection interface.

- A **pointer** type has pointer semantics (just like a C pointer).
- A **raw** type contains untyped data. It's the equivalent of a void* in C. Types that don't contain *raw* in their name have typed data.
- A **mutable** type allows the mutation of the memory it points to.

If you want raw storage but don't need to interact with C, you can use the ManagedBuffer class to allocate the memory. This is what Swift's collections use under the hood to manage their memory. It consists of a single header value (for storing data such as the number of elements) and contiguous memory for the elements. It also has a capacity property, which isn't the same as the number of actual elements: for example, an Array with a count of 17 might own a buffer with a capacity of 32, meaning that 15 more elements can be added before the Array has to allocate more memory. There's also a variant called ManagedBufferPointer, but it doesn't have many applications outside the standard library and may be removed in the future.

Sometimes you need to do manual memory management. For example, you might want to pass a Swift object to a C function with the goal of retrieving it later. C APIs that use callbacks (function pointers) often take an additional context argument (usually an untyped pointer, or void*) that they pass on to each invocation of the callback in order to work around C's lack of closures. When you call such a function from Swift, it'd be convenient to be able to pass a native Swift object as the context value. However, C doesn't know about Swift's memory management. If the API is synchronous, you can just pass in the Swift object and convert it back from the untyped pointer when you receive it in the callback, and all will be fine (we'll look at this in detail in the next section). However, if the API is asynchronous, you have to manually retain and release that object, because otherwise the Swift runtime may deallocate it once it goes out of scope. In order to do that, there's the Unmanaged type. It has methods for retain and release, as well as initializers that either modify or keep the current retain count.

Pointers

In addition to the OpaquePointerType we've already seen, Swift has eight more pointer types that map to different classes of C pointers.

The base type, UnsafePointer, is similar to a const pointer in C. It's generic over the data type of the memory it points to, so UnsafePointer<Int> corresponds to const int*.

Notice that C makes a difference between const int* (a *mutable pointer* to *immutable data*, i.e. you can't write to the pointed data using this pointer) and int* const (an *immutable pointer*, i.e. you can't change where this pointer points to). UnsafePointer in Swift is equivalent to the former variant. As always, you control the mutability of the pointer itself by declaring the variable with var or let.

You can create an unsafe pointer from one of the other pointer types using an initializer. Swift also supports a special syntax for calling functions that take unsafe pointers. You can pass any *mutable* variable of the correct type to such a function by prefixing it with an ampersand, thereby making it an *in-out expression*:

```
var x = 5
func fetch(p: UnsafePointer<Int>) -> Int {
    return p.pointee
}
fetch(p: &x) // 5
```

This looks exactly like the inout parameters we covered in the functions chapter, and it works in a similar manner — although in this case, nothing is passed back to the caller via this value because the pointer isn't mutable. The pointer that Swift creates behind the scenes and passes to the function is guaranteed to be valid only for the duration of the function call. Don't try to return the pointer from the function and access it after the function has returned — the result is undefined.

There's also a mutable variant, named UnsafeMutablePointer. This struct works just like a regular C pointer; you can dereference the pointer and change the value of the memory, which then gets passed back to the caller via the in-out expression:

```
func increment(p: UnsafeMutablePointer<Int>) {
    p.pointee += 1
}
var y = 0
increment(p: &y)
y // 1
```

Rather than using an in-out expression, you can also allocate memory directly using UnsafeMutablePointer. The rules for allocating memory in Swift are similar to the rules in C: after allocating the memory, you first need to initialize it before you can use it. Once you're done with the pointer, you need to deallocate the memory:

```
// Allocate and initialize memory for two Ints
let z = UnsafeMutablePointer<Int>.allocate(capacity: 2)
z.initialize(to: 42, count: 2)
z.pointee // 42
// Pointer arithmetic:
(z+1).pointee = 43
// Subscripts:
z[1] // 43
```

```
// Deallocate the memory
// If Pointee is a non-trivial type (e.g. a class), you must
// call deinitialize before calling deallocate.
z.deallocate(capacity: 2)
// Don't access pointee after deallocate
```

We saw another example of this in the chapter on strings when we allocated a pointer to an `NSRange` for the purpose of receiving a value back from a Foundation API.

In C APIs, it's also very common to have a pointer to a sequence of bytes with no specific element type (`void*` or `const void*`). The equivalent counterparts in Swift are the `UnsafeMutableRawPointer` and `UnsafeRawPointer` types. C APIs that use `void*` or `const void*` get imported as these types. Unless you really need to operate on raw bytes, you'd usually directly convert these types into `Unsafe[Mutable]Pointer` or other typed variants by using one of their instance methods (such as `assumingMemoryBound(to:)`, `bindMemory(to:)`, or `load(fromByteOffset:as:)`).

Unlike C, Swift uses optionals to distinguish between nullable and non-nullable pointers. Only values with an optional pointer type can represent a null pointer. Under the hood, the memory layout of an `UnsafePointer<T>` and an `Optional<UnsafePointer<T>>` is exactly identical; the compiler is smart enough to map the `.none` case to the all-zeros bit pattern of the null pointer.

Sometimes a C API has an opaque pointer type. For example, in the cmark library, we saw that the type `cmark_node*` gets imported as an `OpaquePointer`. The definition of `cmark_node` isn't exposed in the header, and therefore, we can't access the pointee's memory. You can convert opaque pointers to other pointers using an initializer.

In Swift, we usually use the `Array` type to store a sequence of values contiguously. In C, an array is often returned as a pointer to the first element and an element count. If we want to use such a sequence as a collection, we could turn the sequence into an Array, but that makes a copy of the elements. This is often a good thing (because once they're in an array, the elements are memory managed by the Swift runtime). However, sometimes you don't want to make copies of each element. For those cases, there are the Unsafe[Mutable]BufferPointer types. You initialize them with a pointer to the start element and a count. From then on, you have a (mutable) random-access collection. The buffer pointers make it a lot easier to work with C collections.

Finally, the Unsafe[Mutable]RawBufferPointer types make it easier to work with raw memory as collections (they provide the low-level equivalent to Data and NSData).

> *While pointers require you to manually allocate and free memory, they do perform the standard ARC memory management operations on the elements you store through a pointer. When you have an unsafe mutable (buffer) pointer whose Pointee type is a class type, it'll retain every object you store with initialize and release it again when you call deinitialize.*

Function Pointers

Let's look at a concrete example of a C API that uses pointers. Our goal is to write a Swift wrapper for the qsort sorting function in the C standard library. The type as it's imported in Swift's Darwin module (or if you're on Linux, Glibc) is given below:

```
public func qsort(
    _ __base: UnsafeMutableRawPointer!,
    _ __nel: Int,
    _ __width: Int,
    _ __compar: @escaping @convention(c) (UnsafeRawPointer?,
        UnsafeRawPointer?)
    -> Int32)
```

The man page (man qsort) describes how to use the qsort function:

> *The qsort() and heapsort() functions sort an array of nel objects, the initial member of which is pointed to by base. The size of each object is specified by width.*
>
> *The contents of the array base are sorted in ascending order according to a comparison function pointed to by compar, which requires two arguments pointing to the objects being compared.*

And here's a wrapper function that uses qsort to sort an array of Swift strings:

```
func qsortStrings(array: inout [String]) {
    qsort(&array, array.count, MemoryLayout<String>.stride) { a, b in
        let l = a!.assumingMemoryBound(to: String.self).pointee
        let r = b!.assumingMemoryBound(to: String.self).pointee
        if r > l { return -1 }
        else if r == l { return 0 }
        else { return 1 }
    }
}
```

Let's look at each of the arguments being passed to qsort:

- The first argument is a pointer to the base of the array. Swift arrays automatically convert to C-style base pointers when you pass them into a function that takes an UnsafePointer. We have to use the & prefix because it's an UnsafeMutableRawPointer (a void *base in the C declaration). If the function didn't need to mutate its input and were declared in C as const void *base, the ampersand wouldn't be needed. This matches the difference with `inout` arguments in Swift functions.

- Second, we have to provide the number of elements. This one is easy; we can use the `count` property of the array.

- Third, to get the width of each element, we use `MemoryLayout.stride`, *not* `MemoryLayout.size`. In Swift, `MemoryLayout.size` returns the true size of a type, but when locating elements in memory, platform alignment rules may lead to gaps between adjacent elements. The stride is the size of the type, plus some padding (which may be zero) to account for this gap. For strings, size and stride are currently the same on Apple's platforms, but this won't be the case for all types — for example, `MemoryLayout<(Int32, Bool)>.size` is 5 and `MemoryLayout<(Int32, Bool)>.stride` is 8. When translating code from C to Swift, you probably want to write `MemoryLayout.stride` in cases where you would've used `sizeof` in C.

- The last parameter is a pointer to a C function that's used to compare two elements from the array. Swift automatically bridges a Swift function type to a C function pointer, so we can pass any function that has a matching signature. However, there's one big caveat: C function pointers are just pointers; they can't capture any values. For that reason, the compiler will only allow you to provide functions that don't capture any external state (for example, no local variables and no generics). It signifies this with the `@convention(c)` attribute.

The `compar` function accepts two raw pointers. Such an `UnsafeRawPointer` can be a pointer to anything. The reason we have to deal with `UnsafeRawPointer` (and not `UnsafePointer<String>`) is because C doesn't have generics. However, we know that we get passed in a `String`, so we can interpret it as a pointer to a String. We also know the pointers are never nil here, so we can safely force-unwrap them. Finally, the function needs to return an Int32: a positive number if the first element is greater than the second, zero if they're equal, and a negative number if the first is less than the second.

Making It Generic

It's easy enough to create another wrapper that works for a different type of elements; we can copy and paste the code and change String to a different type and we're done. But we should really make the code generic. This is where we hit the limit of C function pointers. The code below fails to compile because it captures things from outside the closure. More specifically, it captures the comparison and equality operators, which are different for each generic parameter. There's nothing we can do about this — we simply encountered an inherent limitation of C:

```
extension Array where Element: Comparable {
    mutating func quicksort() {
        // Error: a C function pointer cannot be formed
        // from a closure that captures generic parameters
        qsort(&self, self.count, MemoryLayout<Element>.stride) { a, b in
            let l = a!.assumingMemoryBound(to: Element.self).pointee
            let r = b!.assumingMemoryBound(to: Element.self).pointee
            if r > l { return -1 }
            else if r == l { return 0 }
            else { return 1 }
        }
    }
}
```

One way to think about this limitation is by thinking like the compiler. A C function pointer is just an address in memory that points to a block of code. For functions that don't have any context, this address will be static and known at compile time. However, in case of a generic function, an extra parameter (the *generic type) is passed in. Therefore, there are no addresses for specialized generic functions. This is the same for closures. Even if the compiler could rewrite a closure in such a way that it'd be possible to pass it as a function pointer, the memory management couldn't be done automatically — there's no way to know when to release the closure.*

In practice, this is a problem for many C programmers as well. On macOS, there's a variant of qsort called qsort_b, which takes a block — a closure — instead of a function pointer as the last parameter. If we replace qsort with qsort_b in the code above, it'll compile and run fine.

However, qsort_b isn't available on most platforms since blocks aren't part of the C standard. And other functions aside from qsort might not have a block-based variant either. Most C APIs that work with callbacks offer a different solution. They take an extra UnsafeRawPointer as a parameter and pass that pointer on to the callback function. The user of the API can then use this to pass an arbitrary piece of data to each invocation of the callback function. qsort also has a variant, qsort_r, which does exactly this. Its type signature includes an extra parameter, thunk, which is an UnsafeRawPointer. Note that this parameter has also been added to the type of the comparison function pointer because qsort_r passes the value to that function on every invocation:

```
public func qsort_r(
    _ __base: UnsafeMutableRawPointer!,
    _ __nel: Int,
    _ __width: Int,
    _ __thunk: UnsafeMutableRawPointer!,
    _ __compar: @escaping @convention(c)
    (UnsafeMutableRawPointer?, UnsafeRawPointer?, UnsafeRawPointer?)
        -> Int32
)
```

If qsort_b isn't available on our platform, we can reconstruct it in Swift using qsort_r. We can pass anything we want as the thunk parameter, as long as we cast it to an UnsafeRawPointer. In our case, we want to pass the comparison closure. We can automatically create an UnsafeRawPointer out of a variable defined with var by using an in-out expression. So all we need to do is store the comparison closure that's passed as an argument to our qsort_b variant in a variable named thunk. Then we can pass the reference to the thunk variable into qsort_r. Inside the callback, we cast the void pointer back to its real type, Block, and then simply call the closure:

```
typealias Block = (UnsafeRawPointer?, UnsafeRawPointer?) -> Int32
func qsort_block(_ array: UnsafeMutableRawPointer, _ count: Int,
                 _ width: Int, f: @escaping Block)
{
    var thunk = f
    qsort_r(array, count, width, &thunk) { (ctx, p1, p2) -> Int32 in
        let comp = ctx!.assumingMemoryBound(to: Block.self).pointee
        return comp(p1, p2)
    }
}
```

Using qsort_block, we can redefine our qsortWrapper function and provide a nice generic interface to the qsort algorithm that's in the C standard library:

```
extension Array where Element: Comparable {
    mutating func quicksort() {
        qsort_block(&self, self.count, MemoryLayout<Element>.stride) { a, b in
            let l = a!.assumingMemoryBound(to: Element.self).pointee
            let r = b!.assumingMemoryBound(to: Element.self).pointee
            if r > l { return -1 }
            else if r == l { return 0 }
            else { return 1 }
        }
    }
}

var x = [3,1,2]
x.quicksort()
x // [1, 2, 3]
```

It might seem like a lot of work to use a sorting algorithm from the C standard library. After all, Swift's built-in sort function is much easier to use, and it's faster in most cases. However, there are many other interesting C APIs out there that we can wrap with a type-safe and generic interface using the same technique.

Recap

Rewriting an existing C library from scratch in Swift is certainly fun, but it may not be the best use of your time (unless you're doing it for learning, which is totally awesome). There's a lot of well-tested C code out there, and throwing it all out would be a huge waste. Swift is great at interfacing with C code, so why not make use of it? Having said that, there's no denying that most C APIs feel very foreign in Swift. Moreover, it's probably not a good idea to spread C constructs like pointers and manual memory management across your entire code base.

Writing a small wrapper that handles the unsafe parts internally and exposes an idiomatic Swift interface — as we did in this chapter for the Markdown library — gives you the best of both worlds: you don't have to reinvent the wheel (i.e. write a complete Markdown parser) and yet it feels 100 percent native to developers using the API.

Final Words

We hope you enjoyed this journey through Swift with us.

Despite its young age, Swift is already a complex language. It'd be a daunting task to cover every aspect of it in one book, let alone expect readers to remember it all.

We chose the topics for this book largely based on our own interests. What did we want to know that was missing from the official documentation? How do things work under the hood? And perhaps even more importantly, why does Swift behave the way it does?

Even if you don't immediately put everything you learned to practical use, we're confident that having a better understanding of your language makes you a more accomplished programmer.

Swift is still changing rapidly. While the era of large-scale source-breaking changes is probably behind us, there are several areas where we expect major enhancements in the coming years:

- The big goal for Swift 5 is achieving **ABI stability** (application binary interface). This mostly affects internal details like memory layouts and calling conventions, but one prerequisite for *ABI* stability that affects everyone is finalizing the standard library *API*. This is why enhancements to the generics system that the standard library can profit from are a major focus for Swift 5.

- An explicit **memory ownership model** will allow developers to annotate function arguments with ownership requirements. The goal is to give the compiler all the information it needs to avoid unnecessary copies when passing values to functions. We're already seeing the first pieces of this in Swift 4 with the introduction of compiler-enforced exclusive memory access.

- Discussions about adding **first-class concurrency support** to Swift are still in their infancy, and this is a project that will take several years to complete. Still, there's a good chance we'll get something like the coroutine-based `async/await` model that's popular in other languages in the not-too-distant future.

If you're interested in shaping how these and other features turn out, remember that Swift is being developed in the open. Consider joining the swift-evolution mailing list and adding your perspective to the discussions.

Finally, we'd like to encourage you to take advantage of the fact that Swift is open source. When you have a question the documentation doesn't answer, the source code can often give you the answer. If you made it this far, you'll have no problem finding your way through the standard library source files. Being able to check how things are implemented in there was a big help for us when writing this book.